Mark3 Realtime Kernel

Generated by Doxygen 1.8.13

Contents

Chapter 1

The Mark3 Realtime Kernel

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.

The RTOS is written using a super portable design that scales to many common processor architectures, including a variety of 8, 16, 32, and 64-bit targets. The flexible CMake-based build system facilitates compiling the kernel, tests, examples, and user-application code for any supported target with a consistent interface.

The API is rich and surprisingly simple – with six function calls, you can set up the kernel, initialize two threads, and start the scheduler.

Written in modern C++, Mark3 makes use of modern language features that improve code quality, reduces duplication, and simplifies API usage. C++ isn't your thing? No problem! there's a complete set of C-language bindings available to facilitate the use of Mark3 in a wider variety of environments.

The Mark3 kernel releases contain zero compiler warnings, zero compiler errors, and have zero unit test failures. The build and test process can be automated through the Mark3-docker project, allowing for easy integration with continuous integration environments. The kernel is also run through static analysis tools, automated profiling and documentation tools.

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

Chapter 2

License

2.1 License

```
Copyright (c) 2012 - 2019, m0slevin
All rights reserved.

Redistribution and use in source and binary forms, with or without
modification, are permitted provided that the following conditions are met:
    - Redistributions of source code must retain the above copyright
      notice, this list of conditions and the following disclaimer.
    - Redistributions in binary form must reproduce the above copyright
      notice, this list of conditions and the following disclaimer in the
      documentation and/or other materials provided with the distribution.
    - Neither the names m0slevin, Mark Slevinsky, nor the names of its
      contributors may be used to endorse or promote products derived
      from this software without specific prior written permission.
    .

THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS IS" AND
ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED
WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE
DISCLAIMED. IN NO EVENT SHALL FUNKENSTEIN SOFTWARE AND/OR ITS CONTRIBUTORS BE
LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL
DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR
SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED
AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT
(INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS
SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.
```

Chapter 3

Configuring The Kernel

3.1 Overview

Configuration is done through setting options in mark3cfg.h , and portcfg.h.

mark3cfg.h contains global kernel configuration options, which determine specific kernel behaviors, and enable certain kernel APIs independent of the any target architecture. Previous to the R7 release, all kernel configuration options were set from mark3cfg.h, and there was an incredible amount of granularity in the configuration options.

One main motivating factor behind removing the granular configuration in mark3cfg.h is that it added a ton of #ifdefs throughout the code, which made it look less clean. It was also difficult to maintain since there were too many permutations and combinations of configuration options to reasonably test.

Another motivation for removing the vast array of configuration options from mark3cfg.h is that there's limited benefit to code size. With the advent of modern compiler optimizations such as section-based garbage collection and link-time optimizations, compilers to a remarkable job of optimizing out unused code. Mark3 supports these optimizations, allowing for nearly the same level of performance benefit as feature specific #ifdefs. In short - you still only pay for what you use, without having to select the features you want ahead of time.

3.1.1 Kernel Configuration Options

Kernel configuration is performed by setting #define's in mark3cfg.h to values of 0 or 1.

KERNEL_DEBUG

When enabled, assert statements within the kernel are checked at runtime. This is useful for tracking kernel-breaking changes, memory corruption, etc. in debug builds.

Should be disabled in release builds for performance reasons.

KERNEL_STACK_CHECK

Perform stack-depth checks on threads at each context switch, which is useful in detecting stack overflows / near overflows. Near-overflow detection uses thresholds defined in the target's portcfg.h. Enabling this also adds the Thread::GetStackSlack() method, which allows a thread's stack to be profiled on-demand.

Note: When enabled, the additional stack checks result in a performance hit to context switches and thread initialization.

KERNEL_NAMED_THREADS

Enabling this provides the Thread::SetName() and Thread::GetName() methods, allowing for each thread to be named with a null-terminated const char* string.

Note: the string passed to Thread::SetName() must persist for the lifetime of the thread

KERNEL_EVENT_FLAGS

This flag enables the event-flags synchronization object. This feature allows threads to be blocked, waiting on specific condition bits to be set or cleared on an EventFlag object.

While other synchronization objects are enabled by default, this one is configurable because it impacts the Thread object's member data.

KERNEL_CONTEXT_SWITCH_CALLOUT

When enabled, this feature allows a user to define a callback to be executed whenever a context switch occurs. Enabling this provides a means for a user to track thread statistics, but it does result in additional overhead during a context switch.

KERNEL_THREAD_CREATE_CALLOUT

This feature provides a user-defined kernel callback that is executed whenever a thread is started.

KERNEL_THREAD_EXIT_CALLOUT

This feature provides a user-defined kernel callback that is executed whenever a thread is terminated.

KERNEL_ROUND_ROBIN

Enable round-robin scheduling within each priority level. When selected, this results in a small performance hit during context switching and in the system tick handler, as a special software timer is used to manage the running thread's quantum. Can be disabled to optimize performance if not required.

KERNEL_EXTENDED_CONTEXT

Provide a special data pointer in the thread object, which may be used to add additional context to a thread. Typically this would be used to implement thread-local-storage.

3.1.2 Port Configuration Options

The bulk of kernel configuration options reside in the target's portcfg.h file. These options determine various sizes, priorities, and default values related to registers, timers, and threads. Some ports may define their own configuration options used locally by its kerneltimer/kernelswi/threadport modules; these are not shown here. The common configuration options are described below.

KERNEL_NUM_PRIORITIES

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.

KERNEL_TIMERS_THREAD_PRIORITY

Define the priority at which the kernel timer thread runs. Typically, this needs to be one of the highest

Note: Other threads at or above the timer thread's priority are not permitted to use the kernel's Timer API, as the thread relies on strict priority scheduling to manage the global timer list without requiring additional/excessive critical sections.

THREAD_QUANTUM_DEFAULT

Number of milliseconds to set as the default epoch for round-robin scheduling when multiple ready threads are active within the same priority.

KERNEL_STACK_GUARD_DEFAULT

Set the minimum number of words of margin that each thread's stack must maintain. If a thread's stack grows into theis margin, a kernel assert will be generated in debug builds. This is useful for ensuring that threads are not in danger of overflowing their stacks during development/verification.

K_WORD

Define the size of a data word (in bytes) on the target

K_ADDR

Define the size of an address (in bytes) on the target. This typically only differs from K_WORD on Harvard-architecture CPUs.

K_INT

Define a type to be used as an integer by the kernel.

PORT_PRIO_TYPE

Set a base datatype used to represent each element of the scheduler's priority bitmap.

PORT_PRIO_MAP_WORD_SIZE

Size of PORT_PRIO_TYPE in bytes

PORT_SYSTEM_FREQ

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.

PORT_TIMER_FREQ

Set the timer tick frequency. This is the frequency at which the fixed-frequency kernel tick interrupt occurs.

PORT_KERNEL_DEFAULT_STACK_SIZE

Define the default thread stack size (in bytes)

PORT_KERNEL_TIMERS_THREAD_STACK

Define the Timer thread's stack size (in bytes)

PORT_TIMER_COUNT_TYPE

Define the native type corresponding to the target timer hardware's counter register.

PORT_MIN_TIMER_TICKS

Minimum number of timer ticks for any delay or sleep, required to ensure that a timer cannot be initialized to a negative value.

PORT_OVERLOAD_NEW

Set this to 1 to overload the system's New/Free functions with the kernel's allocator functions. A user must configure the Kernel's allocator functions to point to a real heap implementation backed with real memory in order to use dynamic object creation.

PORT_STACK_GROWS_DOWN

Set this to 1 if the stack grows down in the target architecture, or 0 if the stack grows up

PORT_USE_HW_CLZ

Set this to 1 if the target CPU/toolchain supports an optimized Count-leading-zeros instruction, or count-leading-zeros intrinsic. If such functionality is not available, a general-purpose implementation will be used.

See also

portcfg.h
mark3cfg.h

Chapter 4

Building The Kernel

Mark3 contains its own build system and support scripts, based on CMake and Ninja.

4.1 Prerequisites

To build via CMake, a user requires a suitable, supported toolchain (i.e. gcc-avr, arm-none-eabi-gcc), CMake 3.4.2 or higher, and a backend supported by CMake (i.e. Ninja build).

For example, on debian-based distributions, such as Ubuntu, the avr toolchain can be installed using:

```
apt-get install avr-libc gcc-avr cmake ninja-build
```

Once a sane build environment has been created, the kernel, libraries, examples and tests can be built by running `ninja` from the target's build directory (/out/<target>/). By default, Mark3 builds for the atmega328p target, but the target can be selected by manually configuring environment variables, or by running the ./build/set_target.sh script as follows:

```
./build/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)
```

This script is a thin wrapper for the cmake configuration commands, and clears the output directory before re-initializing cmake for the selected target.

4.2 Building

To build the Mark3 kernel and middleware libraries for a generic ARM Cortex-M0 using a pre-configured arm-none-eabi-gcc toolchain, one would run the following commands:

```
./build/set_target.sh cm0 generic gcc
./build/build.sh
```

To perform an incremental build, go into the target's cmake build directory (/out/<target/) and simply run 'ninja'.

To perform a clean build, go into the target's cmake build directory (/out/<target/) and run 'ninja clean', then 'ninja'.

Note that not all libraries/tests/examples will build in all kernel configurations. The default kernel configuration may need adjustment/tweaking to support a specific part. See the documentation for details.

4.3 Supported Targets

Currently, Mark3 supports GCC toolchains for the following targets:

```
atmega328p
atmega644
atmega1284p
atxmega256a3
atmega1280
atmega2560
msp430f2274
ARM Cortex-M0
ARM Cortex-M3 (Note: Also supports Cortex-M4)
ARM Cortex-M4F (Note: Also supports Cortex-M7)
```

4.4 More Info

The Mark3 project also has a ready-made docker image to simplify the process. Please see the Mark3 Docker project here for more details:

`https://github.com/moslevin/mark3-docker`

Chapter 5

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 - in bytes
#define STACK_SIZE_APP      (192)
#define STACK_SIZE_IDLE     (128)

static K_WORD awAppStack[STACK_SIZE_APP / sizeof(K_WORD)];
static K_WORD awIdleStack[STACK_SIZE_IDLE / sizeof(K_WORD)];

//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( awAppStack,            // Pointer to the stack
                    sizeof(awAppStack),      // Size of the stack
                    1,                       // Thread priority
                    (void*)AppEntry,         // Entry function
                    nullptr );               // Entry function argument

    IdleThread.Init( awIdleStack,           // Pointer to the stack
                     sizeof(awIdleStack),    // Size of the stack
                     0,                      // Thread priority
                     (void*)IdleEntry,       // Entry function
                     nullptr );              // 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;
K_WORD awStack[192];

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

...
{
    clMyThread.Init(awStack,            // Pointer to the stack to use by this thread
                    sizeof(awStack),    // Size of the stack in bytes
                    1,                  // Thread priority (0 = idle, 7 = max)
                    (void*)AppEntry,    // Function where the thread starts executing
                    nullptr );          // 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 a timer thread, as a result of a semaphore posted in a timer interrupt. Timer callbacks are executed from a high-priority thread – typically at the highest priority thread in the system. 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 lower-priroity 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 artificically prevent progress from being made.

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

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

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

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

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

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

    clMutex.Release();
}

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

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

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

    clMutex.Release();
}
```

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, EventFlagOperation::Any_Set);

        // 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 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.

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

5.9.1 Notification Example

```
 NOTIFIER

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 Condition Variables

Condition Variables, implemented in Mark3 with the ConditionVariable class, provide an implementation of the classic Monitor pattern. This object allows a thread to wait for a specific condition to occur, claiming a shared lock once the condition is met. Threads may also choose to signal a single blocking thread to indicate a condition has changed, or broadcast condition changes to all waiting threads.

5.10.1 Condition Variable Example

```
// Define a condition variable object, a shared lock, and
// a piece of common data shared between threads to represent
// a condition.

// Assume Mutex and ConditionVariable are initialized
// prior to use.
static ConditionVariable clCondVar;
static Mutex clSharedLock;
static volatile int iCondition = 0;

...

void CondThread1(void *unused_)
{
    while (1)
    {
        // Wait until
        clCondVar.Wait(&clSharedLock);

        // Only act on a specific condition
        if (iCondition == 1337) {
            // Do something
        }

        clSharedLock.Release();
    }
}

void CondThread2(void *unused_)
{
    // Assume Mutex and ConditionVariable are initialized
    // prior to use.
    while (1)
    {
        // Wait until
        clCondVar.Wait(&clSharedLock);

        // Act on a *different* condition than the other thread
        if (iCondition == 5454) {
            // Do something
        }

        clSharedLock.Release();
    }
}

void SignalThread(void* unused)
{
    while (1) {
        // Sleep for a while
        Thread::Sleep(100);

        // Update the condition in a thread-safe manner
        clSharedLock.Claim();
        iCondition = 1337;
```

```
        clSharedLock.Release();

        // Wake one thread to check for the updated condition
        clCondVar.Signal();

        // Sleep for a while
        Thread::Sleep(100);

        // Update the condition in a thread-safe manner
        clSharedLock.Claim();
        iCondition = 1337;
        clSharedLock.Release();

        // Wake all threads to check for the updated condition
        clCondVar.Broadcast();
    }
}
```

5.11 Reader-Write Locks

Reader-Writer locks are provided in Mark3 to provide an efficient way for multiple threads to share concurrent, non-destructive access to a resource, while preventing concurrent destructive/non-destructive accesses. A single "writer" may hold the lock, or 1-or-more "readers" may hold the lock. In the case that readers hold the lock, writers will block until all readers have relinquished their access to the resource. In the case that a writer holds the lock, all other readers and writers must wait until the lock is relinquished.

5.11.1 Reader-Write Lock Example

```
void WriterTask(void* param)
{
    auto pclRWLock = static_cast<ReaderWriterLock*>(param);

    pclRWLock->AcquireWriter();
    // All other readers and writers will have to wait until
    // the lock is released.
    iNumWrites++;
    ...
    pclRWLock->ReleaseWriter();
}

void ReaderTask(void* param)
{
    auto pclRWLock = static_cast<ReaderWriterLock*>(param);

    pclRWLock->AcquireReader();
    // Any number of reader threads can also acquire the lock
    // without having to block, waiting for this task to release it.
    // Writers must block until all readers have released their references
    // to the lock.
    iNumReads++;
    ...
    pclRWLock->ReleaseReader();
}
```

5.12 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;
}
```

5.13 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.

5.14 Coroutines

Mark3 implements a coroutine scheduler, capable of managing a set of prioritized run-to-completion tasks. This is a simple and lightweight cooperative scheduling mechanism, that trades the preemption and synchonization capabilities of threads for simplicity. It is an ideal mechanism to use for background processes in a system, or for performing a coordinating a group of tasks where the relative priority of task execution is important, but the duration of individual tasks is less important.

Like the Mark3 thread scheduler, the coroutine scheduler supports multiple priorities of tasks. Multiple coroutines activated at the same priority level are executed in first-in first-out order.

Coroutines are activated by interrupts, threads, or from within other co-routines. Once activated, the co-routine is able to be scheuduled.

The coroutine scheduler is called by the application from a thread priority. So long as there are activated tasks to be scheduled, the scheduler will return a pointer to the highest priority active coroutine to be run.

Running a co-routine de-activates the co-routine, meaning that coroutines must be re-activated every time they are run.

```
Coroutine cr1;
Coroutine cr2;
Coroutine cr3;

void CoRoutineInit()
{
    CoScheduler::Init();

    cr1.Init(<priority>, <handler function>, <data passed to handler function>);
    cr2.Init(<priority>, <handler function>, <data passed to handler function>);
    cr3.Init(<priority>, <handler function>, <data passed to handler function>);
}

void AsyncEvent1()
{
    // Some event occurred requiring us to run cr1.  Activating a task does not
    // cause the coroutine to be run immediately, but schedules it to be returned by
```

```
    // CoScheduler::Schedule() when there are no higher-priroity active tasks to
    // run.
    cr1.Activate();
}

void AsyncEvent2()
{
    // Some event occurred requiring us to run cr2.
    cr2.Activate();
}

void AsyncEvent3()
{
    // Some event occurred requiring us to run cr3.
    cr3.Activate();
}

...
void CoRoutineSchedulerTask()
{
    while (1) {
        auto* coroutineToRun = CoScheduler::Schedule();
        if (coroutineToRun) {
            // Scheduled task is removed from the scheduler once run.  It must
            // be reactivated to be run again
            coroutineToRun->Run();
        }
        // If run from idle thread, could go into sleep mode instead of running in
        // a tight loop.
    }
}
```

5.15 Critical Guards

Often times, it is useful in a real-time multi-threaded system to place a critical section around a block of code to protect it against concurrent access, or to protect global data from access from interrupts. In Mark3 there are a few different ways of implementing critical sections. Historically, the CS_ENTER() and CS_EXIT() macros were used for this purpose; however, the usage of matching entry/exit macros can often-times be cumbersome and error-prone.

The CriticalGuard object allows a user to wrap a block of code in a critical section, where the critical section is entered when the critical guard object is declared, and the critical section is exited when the object goes out-of-scope.

It is essentially an RAII-style critical section object, that provides the benefit of critical sections without the hassle of having to carefully match enter/exit statements.

```
void MyFunc()
{
    // operations outside of critical section
    {
        auto cg = CriticalGuard{};
        // operations protected by critical section
        // critical section ends when CriticalGuard object goes out of scope
    }
    // Operations outside of critical section
}
```

5.16 Lock Guards

The modern C++ standard library provides RAII-style mutex locking. Unfortunately such an implementation is not usable in the context of Mark3, because the STL implementation supplied for an embedded C++ toolchain would not be aware of Mark3's threading model or synchronization primatives. Mark3 provides its own RAII mutex locking mechanism in the form of LockGuard objects. When a LockGuard object is declared (referencing a valid and initialized mutex object at construction), the lock is claimed upon declaration, and released when the object goes out-of-scope.

```
Mutex clMutex;
void MyFunc()
{
    // operations outside of mutex-locked context
    {
        auto lg = LockGuard{&clMutex};
        // operations inside mutex-locked context
        // mutex automatically unlocked when LockGuard goes out of scope
    }
    // Operations outside of mutex-locked context
}
```

5.17 Scheduler Guards

Similar to the LockGuard and CriticalGuard objects, the SchedulerGuard object provides a coped scheduler-disabled blocks. This essentially gives the executing thread exclusive control of the CPU - except for interrupts - for the duration of the block wrapped in the SchedulerGuard. The scheduler is disabled when the object is declared, and scheduler state is restored when the SchedulerGuard object goes out-of-scope. This is yet another form of RAII-based resource locking in Mark3.

```
void MyFunc()
{
    // thread scheduler enabled
    {
        auto sg = SchedulerGuard{};
        // thread-safety guaranteed - scheduler disabled
        // thread scheduler re-enabled when SchedulerGuard object goes out of scope
    }
    // thread scheduler enabled
}
```

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

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

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

Finally, I wanted to prove that a kernel written entirely in C++ could perform just as well as one written in C. 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 to 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:

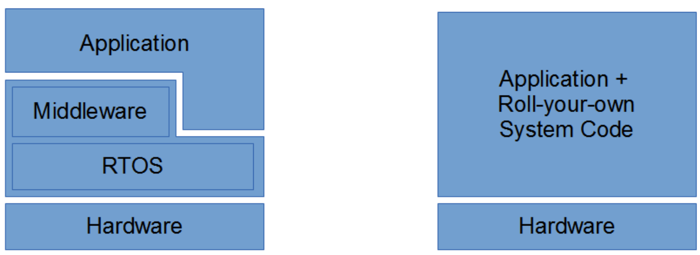


Figure 7.1 Arch

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 1kB 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, dynamic timers, efficient message passing, and multiple types of synchronization primatives.

9.1.4 No external dependencies, no non-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, and libraries (C standard, STL, boost, 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.

9.1.5 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.

9.1.6 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.

9.1.7 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.

9.1.8 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.

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:

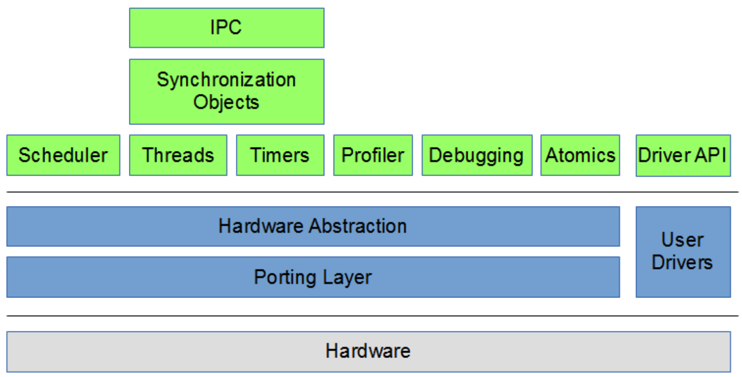


Figure 10.1 Overview

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:

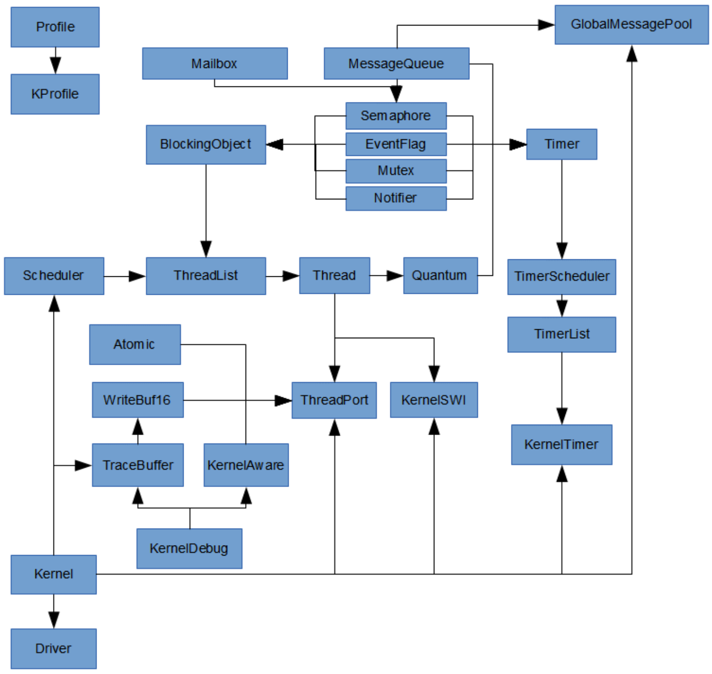


Figure 10.2 Overview

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.

Atomic Operations
Atomic - atomic.cpp/.h

Memory Allocators
AutoAlloc - autoalloc.cpp/.h

CoRoutines
CoScheduler - cosched.cpp/.h
CoList - colist.cpp/.h
Coroutine - coroutine.cpp/.h

Data Structures
LinkList - ll.cpp/.h
PriorityMapL1 - priomapl1.h
PriorityMapL2 - priomapl2.h

Threads + Scheduling
Thread - thread.cpp/.h
Scheduler - scheduler.cpp/.h
Quantum - quantum.cpp/.h
ThreadPort - threadport.cpp/.h $**$
KernelSWI - kernelswi.cpp/.h $**$
ThreadList - threadlist.cpp/.h
ThreadListList - threadlistlist.cpp/.h

Profiling
ProfileTimer - profile.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
ConditionVariable - condvar.cpp/.h
ReaderWriterLock - readerwriter.cpp/.h
CriticalGuard - criticalguard.h
CriticalSection - criticalsection.h
LockGuard - lockguard.h
SchedGuard - schedguard.h

IPC/Message-passing
Mailbox - mailbox.cpp/.h
MessageQueue - message.cpp/.h

Debugging
Miscellaneous Macros - kerneldebug.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:

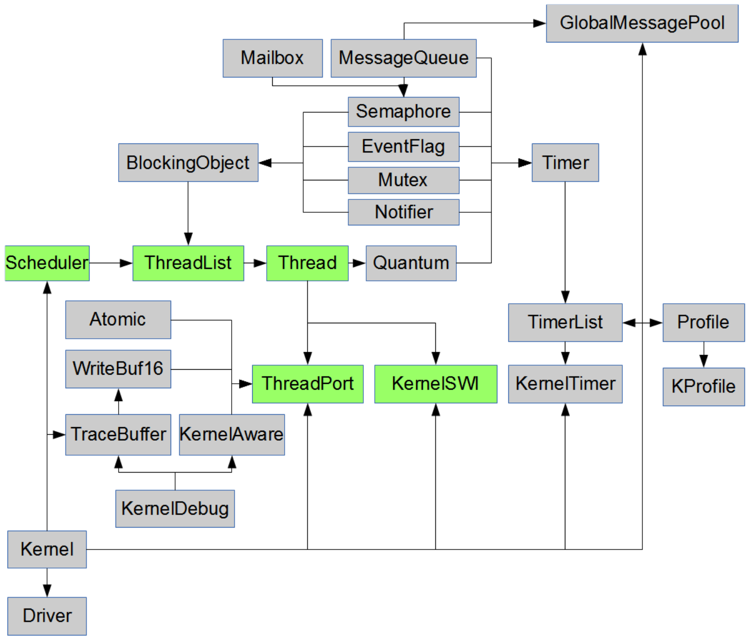


Figure 10.3 Threads and Scheduling

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 primatives (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 primatives 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:

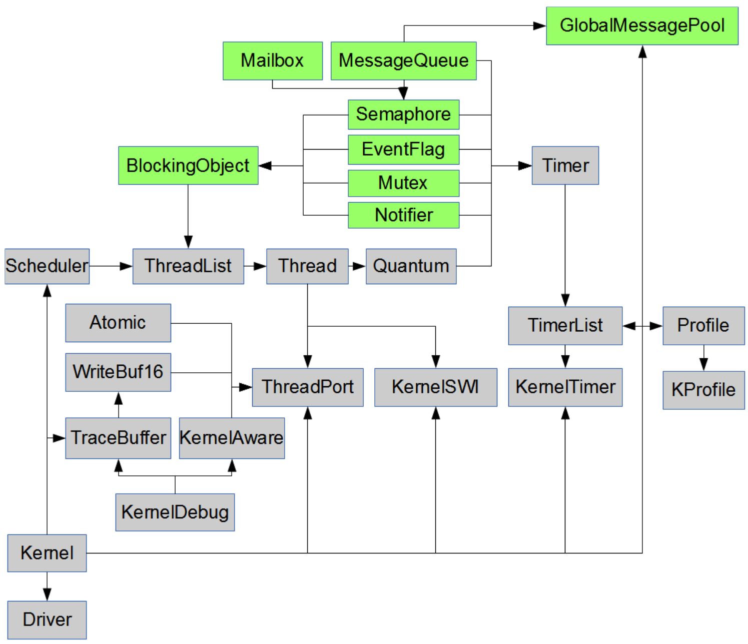


Figure 10.4 Blocking Objects

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 Thread.
Select the next thread to run as the first thread in that list
```

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() != nullptr)
    {
        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 use all approaches. In the event a target architecture or toolchain has intrinsic support for a count-leading-zeroes (PORT_CLZ) instruction, that implementation is used. Otherwise, a software-based implementation is provided – either using a lookup table, or a bitshift-and-compare algorithm.

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

For the lookup-table approach - a 4-bit lookup table can be used with an 8-bit priority-level bitmap would look 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 an optimized 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 Threads in the priority of the next ready thread.

If there are multiple threads within that priority, the kernel starts a one-shot software timer which is programmed to expire at the next Thread's configured quantum. When this timer expires, a timer 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 to 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:

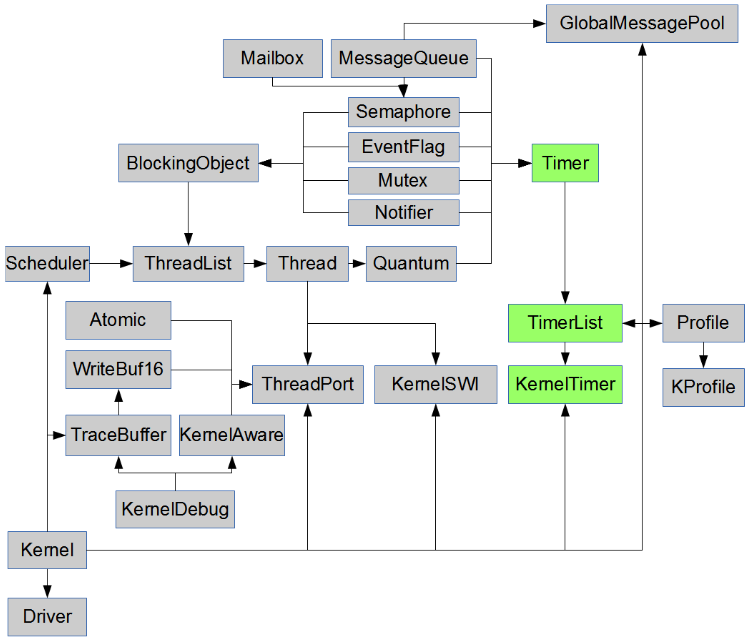


Figure 10.5 Timers

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

Note: Tickless timers are removed as of the R7 release. The below documentation is preserved for historical information only. The reason for removing tickless timers include the overhead associated with managing those timers (significantly more math and management is involved). In practice, there are few scenarios where purely tickless timers add benefit - beyond the most constrained devices. Also, it is entirely possible to disable software timers from the idle task when lower power/fewer interrupts are desired in most cases where a tickless timer would be of value.

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 $\sim$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

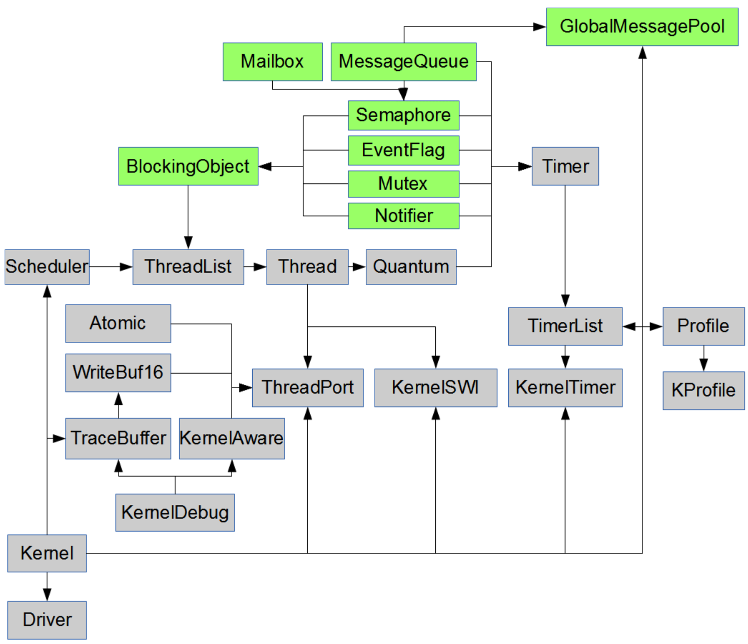


Figure 10.6 Synchronization and IPC

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 Message Queues

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 a source 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 source 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 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.

10.7.4 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

10.7.5 Atomic Operations

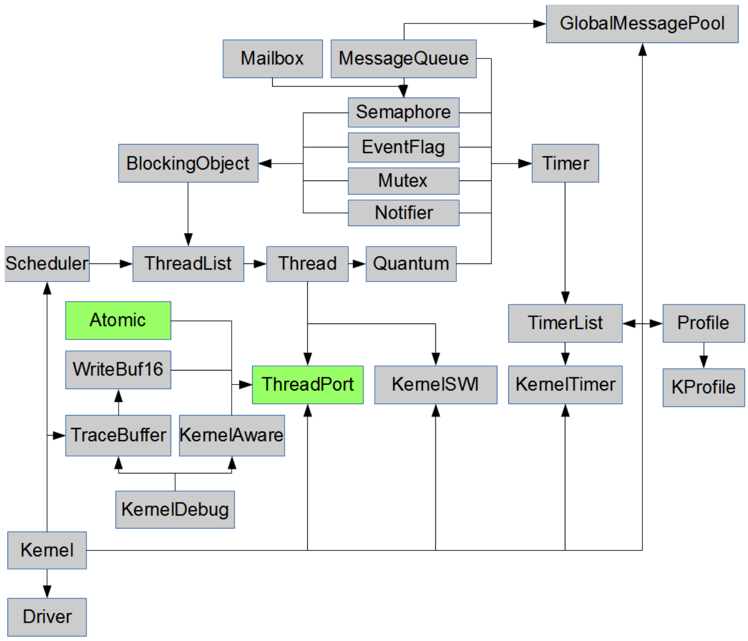


Figure 10.7 Atomic operations

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

10.7.6 Drivers

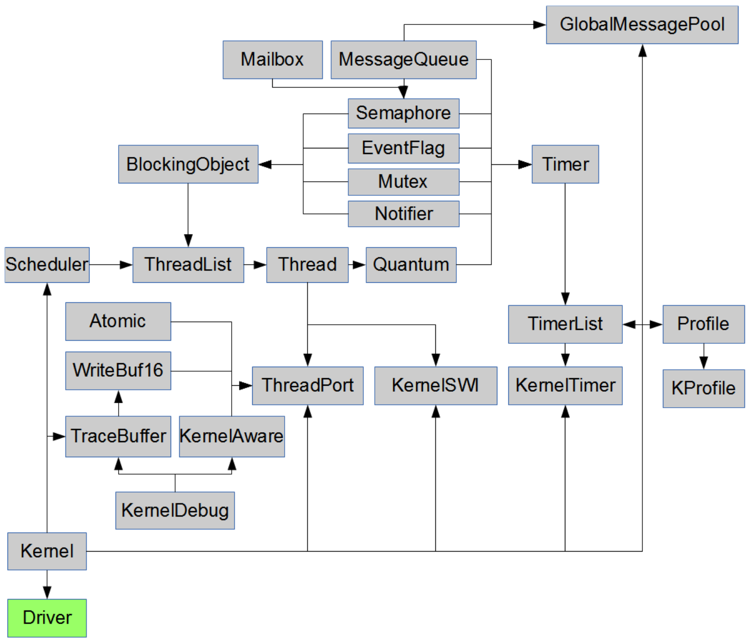


Figure 10.8 Drivers

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::FindByName("/dev/name"). This function returns a pointer to the specified driver if successful, or to a built in /dev/null device if the path name is invalid. After a driver is open, that pointer is used for all other driver access functions.

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

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

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

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

10.8 Kernel Proper and Porting

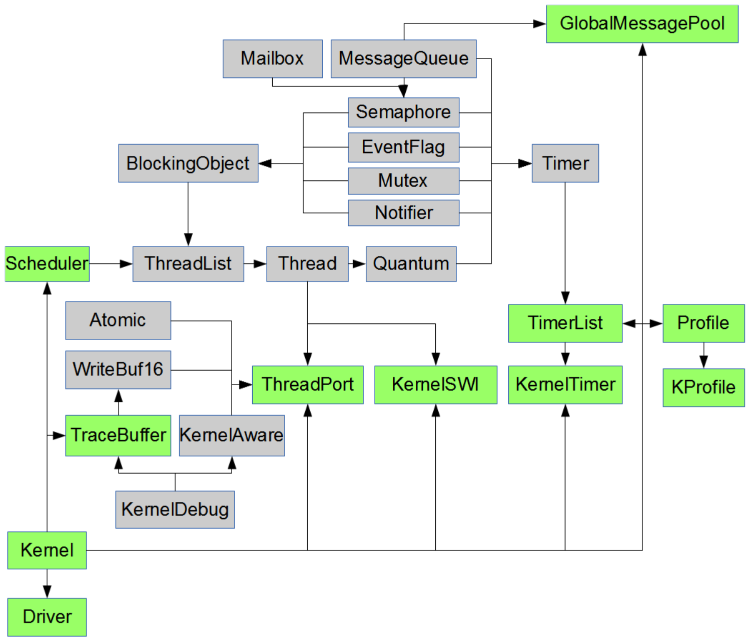


Figure 10.9 Kernel Proper and Porting

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.

```
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;
    }

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

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

    //-- Simulated Manually-Stacked Registers --
    PORT_PUSH_TO_STACK(pulStack, 0x11);
    PORT_PUSH_TO_STACK(pulStack, 0x10);
    PORT_PUSH_TO_STACK(pulStack, 0x09);
    PORT_PUSH_TO_STACK(pulStack, 0x08);
    PORT_PUSH_TO_STACK(pulStack, 0x07);
    PORT_PUSH_TO_STACK(pulStack, 0x06);
    PORT_PUSH_TO_STACK(pulStack, 0x05);
    PORT_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:

@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 (CriticalSection::Enter() and CriticalSection::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, CriticalSection::Enter() performs the following tasks:

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

Conversely, CriticalSection::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 CriticalSection::Enter()    \
{    \
    K_ULONG __ulRegState;    \
    asm    ( \
    " mrs r0, PRIMASK \n"    \
    " mov %[STATUS], r0 \n" \
    " cpsid i \n "    \
    : [STATUS] "=r" (__ulRegState) \
    );

//------------------------------------------------------------------------
#define CriticalSection::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

C-language bindings

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.

11.1 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_)
```

11.2 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.

Chapter 12

Release Notes

12.1 R10 Release

- New: Coroutines + Cooperative scheduler
- New: Critical section APIs defined in kernel lib
- New: RAII critical section (CriticalGuard object)
- New: RAII scheduler-disabled context (SchedulerGuard object)
- Kernel code updated to use RAII critical sections instead of CS_ENTER/CS_EXIT macros
- Updated documentation

12.2 R9 Release

- New: templated linked-lists to avoid explicit casting used in list traversal
- New: ThreadListList class to efficiently track all threads in the system
- Remove use of C-style casts in kernel
- Remove use of 0 as nullptr in kernel
- Refactor code to use constexpr instead of C-style preprocessor defines where possible
- Refactor priority-map class as a set of template classes, reducing use of macros and defines
- Fix a "disappearing thread" bug where an inopportune context switch could cause a thread to get lost
- Docs no longer build by default

12.3 R8 Release

- Structural changes to separate the kernel from the rest of Mark3-repo
- Cleanup and reformatting

12.4 R7 (Full Throttle) Release

- Re-focusing project on kernel, integrating with 3rd party code instead of 1st party middleware
- Re-focusing on atmega1284p and cortex-m as default targets
- New: Refactored codebase to C++14 standard
- New: Moved non-kernel code, drivers, libs, and BSPs to separate repos from kernel
- New: Modular repository-based structure, managed via Android's Repo tool
- New: ConditionVariable kernel API
- New: ReaderWriterLock kernel API
- New: AutoAlloc redirects to user-defined allocators
- New: Global new() and delete() overrides redirect to AutoAlloc APIs
- New: RAII Mutex Locking APIs
- New: Support for cortex-a53 (aarch64) targets
- New: Doxygen builds as part of cmake process
- Updated Mark3c for new APIs
- Moved driver layer out of the kernel
- Moved all non-essential libraries out of the kernel (into other repos)
- Build system supports modular BSP architecture
- Removed fake idle-thread feature, since it doesn't support all targets
- Unit tests and examples will run on any target with a BSP
- Moved AVR-specific code out of the kernel (kernelaware debugging support)
- Remove most build-time configuration flags from mark3cfg.h, and remove associated ifdefs throughout the code.
- Support qemu-system-arm's lm3s6965 evb target, with semihosting
- Support qemu-system-arm's rasbpi3 evb target, with semihosting
- Incrase test coverage
- Various bugfixes and improvements

12.5 R6 Release

- New: Replace recursive-make build system with CMake and Ninja
- New: Transitioned version control to Git from Subversion.
- New: Socket library, implementing named "domain-socket" style IPC
- New: State Machine framework library
- New: Software I2C library completed, with demo app
- New: Kernel Timer loop can optionally be run within its own thread instead of a nested interrupt

- New: UART drivers are all now abstracted throught UartDriver base class for portability
- Experimental: Process library, allowing for the creation of resource-isolated processes
- Removed: Bare-metal support for Atmel SAMD20 (generic port still works)
- Cleanup all compiler warnings on atmega328p
- Various Bugfixes and optimizations
- Various Script changes related to automating the build + release process

12.6 R5 Release

- New: Shell library for creating responsive CLIs for embedded applications (M3Shell)
- New: Stream library for creating thread-safe buffered streams (streamer)
- New: Blocking UART implementation for AVR (drvUARTplus)
- New: "Extended context" kernel feature, which is used to implement thread-local storage
- New: "Extra Checks" kernel feature, which enforces safe API usage under pain of Kernel Panic
- New: Realtime clock library
- New: Example application + bsp for the open-hardware Mark3no development board (mark3no)
- New: Kernel objects descoped/destroyed while still in active use will now cause kernel panic
- New: Kernel callouts for thread creation/destruction/context switching, used for time tracking
- New: Simple power management class
- New: WIP software-based I2C + SPI drivers
- Optimized thread scheduling via target-optimized "count-leading-zero" macros
- Expanded memutil library
- Various optimizations of ARM Cortex-M assembly code
- Various bugfixes to Timer code
- Improved stack overflow checking + warning (stack guard kernel feature)
- AVR bootloader now supports targets with more than 64K of flash
- Moved some port configuration out of platform.mak into header files in the kernel port code
- The usual minor bugfixes and "gentle refactoring"

12.7 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

12.8 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

12.9 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.

12.10 R1 - 2nd Release Candidate

- New: Added support for ARM Cortex-M0 targets
- New: Added support for various 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

12.11 R1 - 1st Release Candidate

- Initial release, with support for AVR microcontrollers

Chapter 13

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 and portcfg.h at compile time. Note that these sizes represent the echo 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 atmega1284p-based targets using gcc. Results are not necessarily indicative of relative or absolute performance on other platforms or toolchains.

13.1 Information

Date Profiled: Wed Jun 12 21:37:23 EDT 2019

13.2 Compiler Version

avr-gcc (GCC) 5.4.0 Copyright (C) 2015 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.

13.3 Profiling Results

Mark3 Module Size Report:

```
-  Atomic Operations............................... : 56 Bytes
-  Allocate-once Heap.............................. : 120 Bytes
-  Synchronization Objects - Base Class............ : 126 Bytes
-  Condition Variables (Synchronization Object).... : 286 Bytes
-  Coroutine task-list management.................. : 122 Bytes
-  Main coroutine task object...................... : 336 Bytes
-  Coroutine task scheduler........................ : 318 Bytes
-  Synchronization Object - Event Flag............. : 872 Bytes
-  Mark3 Kernel Base Class......................... : 142 Bytes
-  Semaphore (Synchronization Object).............. : 568 Bytes
-  Fundamental Kernel Linked-List Classes.......... : 560 Bytes
-  RAII Locking Support based on Mark3 Mutex class. : 62 Bytes
-  Mailbox IPC Support............................. : 1064 Bytes
-  Message-based IPC............................... : 288 Bytes
-  Mutex (Synchronization Object).................. : 658 Bytes
```

```
-  Notification Blocking Object.................... : 626 Bytes
-  Performance-profiling timers.................... : 366 Bytes
-  Round-Robin Scheduling Support.................. : 265 Bytes
-  Reader-writer Locks (Synchronization Object).... : 206 Bytes
-  Thread Scheduling............................... : 570 Bytes
-  Thread Implementation........................... : 1501 Bytes
-  Fundamental Kernel Thread-list Data Structures.. : 440 Bytes
-  ThreadListList Data Structures.................. : 20 Bytes
-  Software Timer Kernel Object.................... : 241 Bytes
-  Software Timer Management....................... : 412 Bytes
-  Atmel AVR - Kernel Interrupt Implemenation....... : 28 Bytes
-  Atmel AVR - Kernel Timer Implementation.......... : 810 Bytes
-  Atmel AVR - Basic Threading Support.............. : 506 Bytes
```

Mark3 Kernel Size Summary:

```
- Kernel                    :  3359 Bytes
- Synchronization Objects   :  4630 Bytes
- Port                      :  1344 Bytes
- Features                  :  1460 Bytes
- Coroutines                :  776 Bytes
- Untracked Objects         :  0 Bytes
- Total Size                :  11569 Bytes
```

Chapter 14

Namespace Index

14.1 Namespace List

Here is a list of all namespaces with brief descriptions:

Chapter 15

Hierarchical Index

15.1 Class Hierarchy

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

Chapter 16

Class Index

16.1 Class List

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

Chapter 17

File Index

17.1 File List

Here is a list of all files with brief descriptions:

Chapter 18

Namespace Documentation

18.1 Mark3 Namespace Reference

Namespaces

- Atomic

  The Atomic namespace This utility module provides primatives for atomic operations - that is, operations that are guaranteed to execute uninterrupted. Basic atomic primatives provided here include Set/Add/Subtract, as well as an atomic test-and-set.

Classes

- class AutoAlloc

  The AutoAlloc class. This class provides an object-allocation interface for both kernel objects and user-defined types. This class supplies callouts for alloc/free that use object-type metadata to determine how objects may be allocated, allowing a user to create custom dynamic memory implementations for specific object types and sizes. As a result, the user-defined allocators can avoid the kinds of memory fragmentation and exhaustion issues that occur in typical embedded systems in which a single heap is used to satisfy all allocations in the application.
- class BlockingObject

  The BlockingObject class. 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.
- class CircularLinkList

  The CircularLinkList class Circular-linked-list data type, inherited from the base LinkList type.
- class CoList

  The CoList class The CoList class implements a circular-linked-listed structure for coroutine objects. The intent of this object is to maintain a list of active coroutine objects with a specific priority or state, to ensure that a freshly-schedulable co-routine always exists at the head of the list.
- class ConditionVariable

  The ConditionVariable class This class implements a condition variable. This is a synchronization object that allows multiple threads to block, each waiting for specific signals unique to them. Access to the specified condition is guarded by a mutex that is supplied by the caller. This object can permit multiple waiters that can be unblocked one-at-a-time via signalling, or unblocked all at once via broadcasting. This object is built upon lower-level primatives, and is somewhat more heavyweight than the primative types supplied by the kernel.
- class Coroutine

  The Coroutine class implements a lightweight, run-to-completion task that forms the basis for co-operative task scheduling in Mark3. Coroutines are designed to be run from a singular context, and scheduled as a result of events occurring from threads, timers, interrupt sources, or other co-routines.

- class CoScheduler

  The CoScheduler class. This class implements the coroutine scheduler. Similar to the Mark3 thread scheduler, the highest-priority active object is scheduled / returned for execution. If no active co-routines are available to be scheduled, then the scheduler returns nullptr.

- class CriticalGuard

  The CriticalGuard class. This class provides an implemention of RAII for critical sections. Object creation results in a critical section being invoked. The subsequent destructor call results in the critical section being released.

- class CriticalSection

  The CriticalSection class. This class implements a portable CriticalSection interface based on macros/inline functions that are implemented as part of each port.

- class DoubleLinkList

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

- class EventFlag

  The EventFlag class. This class implements 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.

- class Kernel

  The Kernel Class encapsulates all of the kernel startup, configuration and management functions.

- class KernelSWI

  The KernelSWI Class provides the software-interrupt used to implement the context-switching interrupt used by the kernel. This interface must be implemented by target-specific code in the porting layer.

- class KernelTimer

  The KernelTimer class provides a timer interface used by all time-based scheduling/timer subsystems in the kernel. This interface must be implemented by target-specific code in the porting layer.

- class LinkList

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

- class LinkListNode

  The LinkListNode Class Basic linked-list node data structure. This data is managed by the linked-list class types, and can be used transparently between them.

- class LockGuard

  The LockGuard class. This class provides RAII locks based on Mark3's kernel Mutex object. Note that Mark3 does not support exceptions, so care must be taken to ensure that this object is only used where that constraint can be met.

- class Mailbox

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

- class MemUtil

  String and Memory manipu32ation class.

- class Message

  the Message class. This object provides threadsafe message-based IPC services based on exchange of objects containing a data pointer and minimal application-defined metadata. Messages are to be allocated/produced by the sender, and deallocated/consumed by the receiver.

- class MessagePool

  The MessagePool Class The MessagePool class implements a simple allocator for message objects exchanged between threads. The sender allocates (pop's) messages, then sends them to the receiver. Upon receipt, it is the receiver's responsibility to deallocate (push) the message back to the pool.

- class MessageQueue

  The MessageQueue class. Implements a mechanism used to send/receive data between threads. Allows threads to block, waiting for messages to be sent from other contexts.

- class Mutex

  The Mutex Class. Class providing Mutual-exclusion locks, based on BlockingObject.

- class Notify

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

- class PriorityMapL1

  The PriorityMapL1 class This class implements a priority bitmap data structure. Each bit in the objects internal storage represents a priority. When a bit is set, it indicates that something is scheduled at the bit's corresponding priority, when a bit is clear it indicates that no entities are scheduled at that priority. This object provides the fundamental logic required to implement efficient priority-based scheduling for the thread + coroutine schedulers in the kernel.

- class PriorityMapL2

  The PriorityMapL2 class This class implements a priority bitmap data structure. Each bit in the objects internal storage represents a priority. When a bit is set, it indicates that something is scheduled at the bit's corresponding priority, when a bit is clear it indicates that no entities are scheduled at that priority. This object provides the fundamental logic required to implement efficient priority-based scheduling for the thread + coroutine schedulers in the kernel.

- class ProfileTimer

  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.

- class Quantum

  The Quantum Class. Static-class used to implement Thread quantum functionality, which is fundamental to round-robin thread scheduling.

- class ReaderWriterLock

  The ReaderWriterLock class. This class implements an object that marshalls access to a resource based on the intended usage of the resource. A reader-writer lock permits multiple concurrent read access, or single-writer access to a resource. If the object holds a write lock, other writers, and all readers will block until the writer is finished. If the object holds reader locks, all writers will block until all readers are finished before the first writer can take ownership of the resource. This is based upon lower-level synchronization primatives, and is somewhat more heavyweight than primative synchronization types.

- class Scheduler

  The Scheduler Class. This class provides priority-based round-robin Thread scheduling for all active threads managed by the kernel.

- class SchedulerGuard

  The SchedulerGuard class This class implements RAII-based control of the scheduler's global state. Upon object construction, the scheduler's state is cached locally and the scheduler is disabled (if not already disabled). Upon object destruction, the scheduler's previous state is restored. This object is interrupt-safe, although it has no effect when called from an interrupt given that interrupts are inherently higher-priority than threads.

- class Semaphore

  the Semaphore class provides Binary & Counting semaphore objects, based on BlockingObject base class.

- class Streamer

  The Streamer class. This class implements a circular byte-buffer with thread and interrupt safe methods for writing-to and reading-from the buffer. Objects of this class type are designed to be shared between threads, or between threads and interrupts.

- class Thread

  The Thread Class. This object providing the fundamental thread control data structures and functions that define a single thread of execution in the Mark3 operating system. It is the fundamental data type used to provide multitasking support in the kernel.

- class ThreadList

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

- class ThreadListList

  The ThreadListList class Class used to track all threadlists active in the OS kernel. At any point in time, the list can be traversed to get a complete view of all running, blocked, or stopped threads in the system.

- class ThreadPort

  The ThreadPort Class defines the target-specific functions required by the kernel for threading.

- class Timer

  The Timer Class. This class provides kernel-managed timers, used to provide high-precision delays. Functionality is useful to both user-code, and is used extensively within the kernel and its blocking objects to implement round-robin scheduling, thread sleep, and timeouts. Provides one-shot and periodic timers for use by application code. This object relies on a target-defined hardware timer implementation, which is multiplexed by the kernel's timer scheduler.

- class TimerList

  the TimerList class. This class implements a doubly-linked-list of timer objects.
- class TimerScheduler

  The TimerScheduler Class. This implements a "Static" class used to manage a global list of timers used throughout the system.
- struct Token_t

  Token descriptor struct format.
- class TypedCircularLinkList

  The TypedCircularLinkList Class Circular-linked-list data type, inherited from the base LinkList type, and templated for use with linked-list-node derived data-types.
- class TypedDoubleLinkList

  The TypedDoubleLinkList Class Doubly-linked-list data type, inherited from the base LinkList type, and templated for use with linked-list-node derived data-types.
- class TypedLinkListNode

  The TypedLinkListNode class The TypedLinkListNode class provides a linked-list node type for a specified object type. This can be used with typed link-list data structures to manage lists of objects without having to static-cast between the base type and the derived class.

Typedefs

- using AutoAllocAllocator_t = void ∗(∗)(AutoAllocType eType_, size_t sSize_)
- using AutoAllocFree_t = void(∗)(AutoAllocType eType_, void ∗pvObj_)
- using CoPrioMap = PriorityMapL1< PORT_PRIO_TYPE, PORT_COROUTINE_PRIORITIES >
- using CoroutineHandler = void(∗)(Coroutine ∗pclCaller_, void ∗pvContext_)
- using PanicFunc = void(∗)(uint16_t u16PanicCode_)
- using IdleFunc = void(∗)()
- using ThreadEntryFunc = void(∗)(void ∗pvArg_)
- using PriorityMap = PriorityMapL1< PORT_PRIO_TYPE, KERNEL_NUM_PRIORITIES >
- using ThreadCreateCallout = void(∗)(Thread ∗pclThread_)
- using ThreadExitCallout = void(∗)(Thread ∗pclThread_)
- using ThreadContextCallout = void(∗)(Thread ∗pclThread_)
- using TimerCallback = void(∗)(Thread ∗pclOwner_, void ∗pvData_)

Enumerations

- enum AutoAllocType : uint8_t {
  AutoAllocType::EventFlag, AutoAllocType::MailBox, AutoAllocType::Message, AutoAllocType::MessagePool, AutoAllocType::MessageQueue, AutoAllocType::Mutex, AutoAllocType::Notify, AutoAllocType::Semaphore, AutoAllocType::Thread, AutoAllocType::Timer, AutoAllocType::ConditionVariable, AutoAllocType::Reader←WriterLock,
  AutoAllocType::User, AutoAllocType::Raw = 0xFF }
- enum EventFlagOperation : uint8_t {
  EventFlagOperation::All_Set = 0, EventFlagOperation::Any_Set, EventFlagOperation::All_Clear, EventFlag←Operation::Any_Clear,
  EventFlagOperation::Pending_Unblock }
- enum ThreadState : uint8_t {
  ThreadState::Exit = 0, ThreadState::Ready, ThreadState::Blocked, ThreadState::Stop,
  ThreadState::Invalid }

Functions

- static void KernelTimer_Task (void *unused)
- static void Thread_Switch (void)
- ISR (INT2_vect) __attribute__((signal

    ISR(INT2_vect) SWI using INT2 - used to trigger a context switch.
- uint8_t PORT_CLZ (uint8_t in_)
- void PORT_IRQ_ENABLE ()
- void PORT_IRQ_DISABLE ()
- void PORT_CS_ENTER ()
- void PORT_CS_EXIT ()
- K_WORD PORT_CS_NESTING ()

Variables

- static constexpr auto uMaxTimerTicks = uint32_t { 0x7FFFFFFF }

    Maximum value to set.
- static constexpr auto uTimerTicksInvalid = uint32_t { 0 }
- static constexpr auto uTimerFlagOneShot = uint8_t { 0x01 }

    Timer is one-shot.
- static constexpr auto uTimerFlagActive = uint8_t { 0x02 }

    Timer is currently active.
- static constexpr auto uTimerFlagCallback = uint8_t { 0x04 }

    Timer is pending a callback.
- static constexpr auto uTimerFlagExpired = uint8_t { 0x08 }

    Timer is actually expired.
- naked
- static constexpr auto SR_ = uint8_t{0x3F}
- K_WORD g_kwSFR
- K_WORD g_kwCriticalCount

18.1.1 Typedef Documentation

18.1.1.1 AutoAllocAllocator_t

```
using Mark3::AutoAllocAllocator_t = typedef void* (*)(AutoAllocType eType_, size_t sSize_)
```

Definition at line 53 of file autoalloc.h.

18.1.1.2 AutoAllocFree_t

```
using Mark3::AutoAllocFree_t = typedef void (*)(AutoAllocType eType_, void* pvObj_)
```

Definition at line 54 of file autoalloc.h.

18.1.1.3 CoPrioMap

```
using Mark3::CoPrioMap = typedef PriorityMapL1<PORT_PRIO_TYPE, PORT_COROUTINE_PRIORITIES>
```

Definition at line 30 of file coroutine.h.

18.1.1.4 CoroutineHandler

```
using Mark3::CoroutineHandler = typedef void (*)(Coroutine* pclCaller_, void* pvContext_)
```

Definition at line 40 of file coroutine.h.

18.1.1.5 IdleFunc

```
using Mark3::IdleFunc = typedef void (*)()
```

Function pointer type used to implement the idle function, where support for an idle function (as opposed to an idle thread) exists.

Definition at line 37 of file kerneltypes.h.

18.1.1.6 PanicFunc

```
using Mark3::PanicFunc = typedef void (*)(uint16_t u16PanicCode_)
```

Function pointer type used to implement kernel-panic handlers.

Definition at line 30 of file kerneltypes.h.

18.1.1.7 PriorityMap

```
using Mark3::PriorityMap = typedef PriorityMapL1<PORT_PRIO_TYPE, KERNEL_NUM_PRIORITIES>
```

Definition at line 29 of file priomap.h.

18.1.1.8 ThreadContextCallout

```
using Mark3::ThreadContextCallout = typedef void (*)(Thread* pclThread_)
```

Definition at line 54 of file thread.h.

18.1.1.9 ThreadCreateCallout

```
using Mark3::ThreadCreateCallout = typedef void (*)(Thread* pclThread_)
```

Definition at line 52 of file thread.h.

18.1.1.10 ThreadEntryFunc

```
using Mark3::ThreadEntryFunc = typedef void (*)(void* pvArg_)
```

Function pointer type used for thread entrypoint functions

Definition at line 43 of file kerneltypes.h.

18.1.1.11 ThreadExitCallout

```
using Mark3::ThreadExitCallout = typedef void (*)(Thread* pclThread_)
```

Definition at line 53 of file thread.h.

18.1.1.12 TimerCallback

```
using Mark3::TimerCallback = typedef void (*)(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 50 of file timer.h.

18.1.2 Enumeration Type Documentation

18.1.2.1 AutoAllocType

```
enum Mark3::AutoAllocType : uint8_t [strong]
```

Enumerator

EventFlag	
MailBox	
Message	
MessagePool	
MessageQueue	
Mutex	
Notify	
Semaphore	
Thread	
Timer	
ConditionVariable	
ReaderWriterLock	
User	
Raw	

Definition at line 32 of file autoalloc.h.

18.1.2.2 EventFlagOperation

```
enum Mark3::EventFlagOperation : uint8_t [strong]
```

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

Enumerator

All_Set	Block until all bits in the specified bitmask are set.
Any_Set	Block until any bits in the specified bitmask are set.
All_Clear	Block until all bits in the specified bitmask are cleared.
Any_Clear	Block until any bits in the specified bitmask are cleared.
Pending_Unblock	Special code. Not used by user

Definition at line 50 of file kerneltypes.h.

18.1.2.3 ThreadState

```
enum Mark3::ThreadState : uint8_t [strong]
```

Enumeration representing the different states a thread can exist in

Enumerator

Exit	
Ready	
Blocked	
Stop	
Invalid	

Definition at line 62 of file kerneltypes.h.

18.1.3 Function Documentation

18.1.3.1 ISR()

```
Mark3::ISR (
             INT2_vect   )
```

ISR(INT2 _vect) SWI using INT2 - used to trigger a context switch.

Definition at line 137 of file threadport.cpp.

18.1.3.2 KernelTimer_Task()

```
static void Mark3::KernelTimer_Task (
             void * unused )   [static]
```

Definition at line 44 of file kerneltimer.cpp.

18.1.3.3 PORT_CLZ()

```
uint8_t Mark3::PORT_CLZ (
             uint8_t in_ )   [inline]
```

Definition at line 144 of file threadport.h.

18.1.3.4 PORT_CS_ENTER()

```
void Mark3::PORT_CS_ENTER ( )   [inline]
```

Definition at line 167 of file threadport.h.

18.1.3.5 PORT_CS_EXIT()

```
void Mark3::PORT_CS_EXIT ( )   [inline]
```

Definition at line 178 of file threadport.h.

18.1.3.6 PORT_CS_NESTING()

```
K_WORD Mark3::PORT_CS_NESTING ( )  [inline]
```

Definition at line 187 of file threadport.h.

18.1.3.7 PORT_IRQ_DISABLE()

```
void Mark3::PORT_IRQ_DISABLE ( )  [inline]
```

Definition at line 161 of file threadport.h.

18.1.3.8 PORT_IRQ_ENABLE()

```
void Mark3::PORT_IRQ_ENABLE ( )  [inline]
```

Definition at line 155 of file threadport.h.

18.1.3.9 Thread_Switch()

```
static void Mark3::Thread_Switch (
             void  )  [static]
```

Definition at line 94 of file threadport.cpp.

18.1.4 Variable Documentation

18.1.4.1 g_kwCriticalCount

```
K_WORD Mark3::g_kwCriticalCount
```

Definition at line 36 of file threadport.cpp.

18.1.4.2 g_kwSFR

```
K_WORD Mark3::g_kwSFR
```

Definition at line 35 of file threadport.cpp.

18.1.4.3 naked

```
Mark3::naked
```

Definition at line 136 of file threadport.cpp.

18.1.4.4 SR_

```
constexpr auto Mark3::SR_ = uint8_t{0x3F}  [static]
```

Definition at line 137 of file threadport.h.

18.1.4.5 uMaxTimerTicks

```
constexpr auto Mark3::uMaxTimerTicks = uint32_t { 0x7FFFFFFF }  [static]
```

Maximum value to set.

Definition at line 32 of file timer.h.

18.1.4.6 uTimerFlagActive

```
constexpr auto Mark3::uTimerFlagActive = uint8_t { 0x02 }  [static]
```

Timer is currently active.

Definition at line 35 of file timer.h.

18.1.4.7 uTimerFlagCallback

```
constexpr auto Mark3::uTimerFlagCallback = uint8_t { 0x04 }  [static]
```

Timer is pending a callback.

Definition at line 36 of file timer.h.

18.1.4.8 uTimerFlagExpired

```
constexpr auto Mark3::uTimerFlagExpired = uint8_t { 0x08 }  [static]
```

Timer is actually expired.

Definition at line 37 of file timer.h.

18.1.4.9 uTimerFlagOneShot

```
constexpr auto Mark3::uTimerFlagOneShot = uint8_t { 0x01 }  [static]
```

Timer is one-shot.

Definition at line 34 of file timer.h.

18.1.4.10 uTimerTicksInvalid

```
constexpr auto Mark3::uTimerTicksInvalid = uint32_t { 0 }  [static]
```

Definition at line 33 of file timer.h.

18.2 Mark3::Atomic Namespace Reference

The Atomic namespace This utility module provides primatives for atomic operations - that is, operations that are guaranteed to execute uninterrupted. Basic atomic primatives provided here include Set/Add/Subtract, as well as an atomic test-and-set.

Functions

- template<typename T >
  T Set (T ∗pSource_, T val_)

  Set Set a variable to a given value in an uninterruptable operation.
- template<typename T >
  T Add (T ∗pSource_, T val_)

  Add Add a value to a variable in an uninterruptable operation.
- template<typename T >
  T Sub (T ∗pSource_, T val_)

  Sub Subtract a value from a variable in an uninterruptable operation.
- bool TestAndSet (bool ∗pbLock)

  TestAndSet Test to see if a variable is set, and set it if is not already set. This is an uninterruptable operation.

18.2.1 Detailed Description

The Atomic namespace This utility module provides primatives for atomic operations - that is, operations that are guaranteed to execute uninterrupted. Basic atomic primatives provided here include Set/Add/Subtract, as well as an atomic test-and-set.

18.2.2 Function Documentation

18.2.2.1 Add()

```
template<typename T >
T Mark3::Atomic::Add (
            T * pSource_,
            T val_ )
```

Add Add a value to a variable in an uninterruptable operation.

Parameters

*pSource_*	Pointer to a variable
*val_*	Value to add to the variable

Returns

Previously-held value in pSource_

Definition at line 64 of file atomic.h.

18.2.2.2 Set()

```
template<typename T >
T Mark3::Atomic::Set (
            T * pSource_,
            T val_ )
```

Set Set a variable to a given value in an uninterruptable operation.

Parameters

*pSource_*	Pointer to a variable to set the value of
*val_*	New value to set in the variable

Returns

Previously-set value

Definition at line 47 of file atomic.h.

18.2.2.3 Sub()

```
template<typename T >
T Mark3::Atomic::Sub (
            T * pSource_,
            T val_ )
```

Sub Subtract a value from a variable in an uninterruptable operation.

Parameters

p← Source← _	Pointer to a variable
*val_*	Value to subtract from the variable

Returns

Previously-held value in pSource_

Definition at line 81 of file atomic.h.

18.2.2.4 TestAndSet()

```
bool Mark3::Atomic::TestAndSet (
            bool * pbLock )
```

TestAndSet Test to see if a variable is set, and set it if is not already set. This is an uninterruptable operation.

If the value is false, set the variable to true, and return the previously-held value.

If the value is already true, return true.

Parameters

pbLock	Pointer to a value to test against. This will always be set to "true" at the end of a call to TestAndSet.

Returns

true - Lock value was "true" on entry, false - Lock was set

Definition at line 26 of file atomic.cpp.

Chapter 19

Class Documentation

19.1 Mark3::AutoAlloc Class Reference

The AutoAlloc class. This class provides an object-allocation interface for both kernel objects and user-defined types. This class supplies callouts for alloc/free that use object-type metadata to determine how objects may be allocated, allowing a user to create custom dynamic memory implementations for specific object types and sizes. As a result, the user-defined allocators can avoid the kinds of memory fragmentation and exhaustion issues that occur in typical embedded systems in which a single heap is used to satisfy all allocations in the application.

```
#include <autoalloc.h>
```

Static Public Member Functions

- static void Init (void)

  Init Initialize the AutoAllocator before use. Called by Kernel::Init().
- static void SetAllocatorFunctions (AutoAllocAllocator_t pfAllocator_, AutoAllocFree_t pfFree_)

  SetAllocatorFunctions Set the functions used by this class to allocate/free memory used in the kernel.
- template<typename T , AutoAllocType e>
  static T ∗ NewObject ()
- template<typename T , AutoAllocType e>
  static void DestroyObject (T ∗pObj_)
- static void ∗ NewUserTypeAllocation (uint8_t eUserType_)

  NewUserTypeAllocation Attempt to allocate a user-defined object type from the heap.
- static void DestroyUserTypeAllocation (uint8_t eUserType_, void ∗pvObj_)

  DestroyUserTypeAllocation Free a previously allocated user-defined object.
- static void ∗ NewRawData (size_t sSize_)

  NewRawData Attempt to allocate a blob of raw data from the heap.
- static void DestroyRawData (void ∗pvData_)

  DestroyRawData Free a previously allocated blob of data allocated via NewRawData()

Static Private Member Functions

- static void ∗ Allocate (AutoAllocType eType_, size_t sSize_)
- static void Free (AutoAllocType eType_, void ∗pvObj_)

Static Private Attributes

- static AutoAllocAllocator_t m_pfAllocator

  Function used to allocate objects.
- static AutoAllocFree_t m_pfFree

  Funciton used to free objectss.

19.1.1 Detailed Description

The AutoAlloc class. This class provides an object-allocation interface for both kernel objects and user-defined types. This class supplies callouts for alloc/free that use object-type metadata to determine how objects may be allocated, allowing a user to create custom dynamic memory implementations for specific object types and sizes. As a result, the user-defined allocators can avoid the kinds of memory fragmentation and exhaustion issues that occur in typical embedded systems in which a single heap is used to satisfy all allocations in the application.

Definition at line 82 of file autoalloc.h.

19.1.2 Member Function Documentation

19.1.2.1 Allocate()

```
void * Mark3::AutoAlloc::Allocate (
            AutoAllocType eType_,
            size_t sSize_ )  [static], [private]
```

Definition at line 58 of file autoalloc.cpp.

19.1.2.2 DestroyObject()

```
template<typename T , AutoAllocType e>
static void Mark3::AutoAlloc::DestroyObject (
            T * pObj_ )  [inline], [static]
```

Template function used to manage the destruction and de-allocation of predefined kernel objects

Definition at line 116 of file autoalloc.h.

19.1.2.3 DestroyRawData()

```
void Mark3::AutoAlloc::DestroyRawData (
            void * pvData_ )  [static]
```

DestroyRawData Free a previously allocated blob of data allocated via NewRawData()

Parameters

*pv↩ Data_*	pointer to previously-created data object

Definition at line 105 of file autoalloc.cpp.

19.1.2.4 DestroyUserTypeAllocation()

```
void Mark3::AutoAlloc::DestroyUserTypeAllocation (
            uint8_t eUserType_,
            void * pvObj_ )  [static]
```

DestroyUserTypeAllocation Free a previously allocated user-defined object.

Parameters

*pvObj_*	Pointer to previously-allocated object, allocated through NewUserTypeAllocation ()
*eUser↩ Type_*	User defined object type, interpreted by the allocator function

Definition at line 95 of file autoalloc.cpp.

19.1.2.5 Free()

```
void Mark3::AutoAlloc::Free (
            AutoAllocType eType_,
            void * pvObj_ )  [static], [private]
```

Definition at line 67 of file autoalloc.cpp.

19.1.2.6 Init()

```
void Mark3::AutoAlloc::Init (
            void )  [static]
```

Init Initialize the AutoAllocator before use. Called by Kernel::Init().

Definition at line 83 of file autoalloc.cpp.

19.1.2.7 NewObject()

```
template<typename T , AutoAllocType e>
static T* Mark3::AutoAlloc::NewObject ( )  [inline], [static]
```

Template function used to manage the allocation of predefined kernel object types

Definition at line 103 of file autoalloc.h.

19.1.2.8 NewRawData()

```
void * Mark3::AutoAlloc::NewRawData (
            size_t sSize_ )  [static]
```

NewRawData Attempt to allocate a blob of raw data from the heap.

Parameters

s↩ Size↩ _	Size of the data blob (in bytes)

Returns

pointer to newly-allocated blob of data, or nullptr on error.

Definition at line 100 of file autoalloc.cpp.

19.1.2.9 NewUserTypeAllocation()

```
void * Mark3::AutoAlloc::NewUserTypeAllocation (
            uint8_t eUserType_ )  [static]
```

NewUserTypeAllocation Attempt to allocate a user-defined object type from the heap.

Parameters

*eUser↩ Type_*	User defined object type, interpreted by the allocator function

Returns

pointer to a newly-created object, or nullptr on error.

Definition at line 90 of file autoalloc.cpp.

19.1.2.10 SetAllocatorFunctions()

```
void Mark3::AutoAlloc::SetAllocatorFunctions (
            AutoAllocAllocator_t pfAllocator_,
            AutoAllocFree_t pfFree_ ) [static]
```

SetAllocatorFunctions Set the functions used by this class to allocate/free memory used in the kernel.

Parameters

*pfAllocator_*	Function to allocate an object based on its type and/or size
*pfFree_*	Function to free a previously-allocated object

Definition at line 76 of file autoalloc.cpp.

19.1.3 Member Data Documentation

19.1.3.1 m_pfAllocator

```
AutoAllocAllocator_t Mark3::AutoAlloc::m_pfAllocator [static], [private]
```

Function used to allocate objects.

Definition at line 157 of file autoalloc.h.

19.1.3.2 m_pfFree

```
AutoAllocFree_t Mark3::AutoAlloc::m_pfFree [static], [private]
```

Funciton used to free objectss.

Funciton used to free objects.

Definition at line 158 of file autoalloc.h.

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

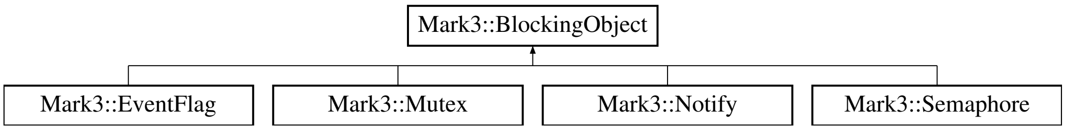
- /home/moslevin/projects/m3-repo/kernel/src/public/autoalloc.h
- /home/moslevin/projects/m3-repo/kernel/src/autoalloc.cpp

19.2 Mark3::BlockingObject Class Reference

The BlockingObject class. 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.

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

Inheritance diagram for Mark3::BlockingObject:



Public Member Functions

- BlockingObject ()
- ∼BlockingObject ()

Protected Member Functions

- void Block (Thread ∗pclThread_)

  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.
- void BlockPriority (Thread ∗pclThread_)

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

  UnBlock Unblock a thread that is already blocked on this object, returning it to the "ready" state by performing the following steps:
- void SetInitialized (void)

  SetInitialized.
- bool IsInitialized (void)

  IsInitialized.

Protected Attributes

- ThreadList m_clBlockList
- uint8_t m_u8Initialized

Static Protected Attributes

- static constexpr auto m_uBlockingInvalidCookie = uint8_t { 0x3C }
- static constexpr auto m_uBlockingInitCookie = uint8_t { 0xC3 }

19.2.1 Detailed Description

The BlockingObject class. 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.

19.2.2 Constructor & Destructor Documentation

19.2.2.1 BlockingObject()

```
Mark3::BlockingObject::BlockingObject ( )  [inline]
```

Definition at line 68 of file blocking.h.

19.2.2.2 ∼BlockingObject()

```
Mark3::BlockingObject::∼BlockingObject ( )  [inline]
```

Definition at line 69 of file blocking.h.

19.2.3 Member Function Documentation

19.2.3.1 Block()

```
void Mark3::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

*pcl↩ Thread_*	Pointer to the thread object that will be blocked.

Definition at line 26 of file blocking.cpp.

19.2.3.2 BlockPriority()

```
void Mark3::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

*pcl↩ Thread_*	Pointer to the Thread to Block.

Definition at line 41 of file blocking.cpp.

19.2.3.3 IsInitialized()

```
bool Mark3::BlockingObject::IsInitialized (
            void )  [inline], [protected]
```

IsInitialized.

Returns

true if initialized, false if object uninitialized

Definition at line 123 of file blocking.h.

19.2.3.4 SetInitialized()

```
void Mark3::BlockingObject::SetInitialized (
            void )  [inline], [protected]
```

SetInitialized.

Definition at line 117 of file blocking.h.

19.2.3.5 UnBlock()

```
void Mark3::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

*pcl←→ Thread_*	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 56 of file blocking.cpp.

19.2.4 Member Data Documentation

19.2.4.1 m_clBlockList

```
ThreadList Mark3::BlockingObject::m_clBlockList [protected]
```

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

Definition at line 133 of file blocking.h.

19.2.4.2 m_u8Initialized

```
uint8_t Mark3::BlockingObject::m_u8Initialized [protected]
```

Token used to check whether or not the object has been initialized prior to use.

Definition at line 139 of file blocking.h.

19.2.4.3 m_uBlockingInitCookie

```
constexpr auto Mark3::BlockingObject::m_uBlockingInitCookie = uint8_t { 0xC3 } [static],
[protected]
```

Definition at line 127 of file blocking.h.

19.2.4.4 m_uBlockingInvalidCookie

```
constexpr auto Mark3::BlockingObject::m_uBlockingInvalidCookie = uint8_t { 0x3C } [static],
[protected]
```

Definition at line 126 of file blocking.h.

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

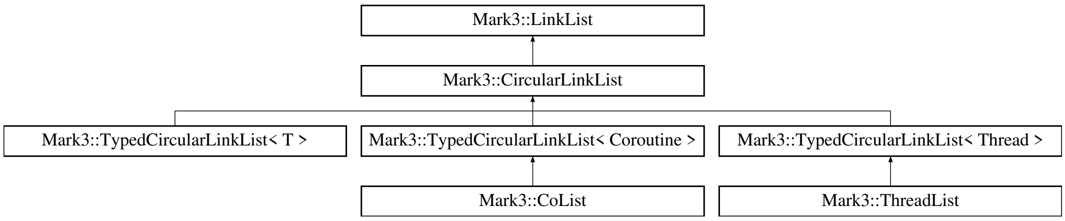
- /home/moslevin/projects/m3-repo/kernel/src/public/blocking.h
- /home/moslevin/projects/m3-repo/kernel/src/blocking.cpp

19.3 Mark3::CircularLinkList Class Reference

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

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

Inheritance diagram for Mark3::CircularLinkList:



Public Member Functions

- void ∗ operator new (size_t sz, void ∗pv)
- CircularLinkList ()
- void Add (LinkListNode ∗node_)

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

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

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

  PivotBackward Pivot the head of the circularly linked list backward (Head = Head->prev, Tail = Tail->prev)
- void InsertNodeBefore (LinkListNode ∗node_, LinkListNode ∗insert_)

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

Additional Inherited Members

19.3.1 Detailed Description

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

Definition at line 218 of file ll.h.

19.3.2 Constructor & Destructor Documentation

19.3.2.1 CircularLinkList()

```
Mark3::CircularLinkList::CircularLinkList ( )  [inline]
```

Definition at line 222 of file ll.h.

19.3.3 Member Function Documentation

19.3.3.1 Add()

```
void Mark3::CircularLinkList::Add (
            LinkListNode * node_ )
```

Add the linked list node to this linked list.

Parameters

*node_*	Pointer to the node to add

Definition at line 81 of file ll.cpp.

19.3.3.2 InsertNodeBefore()

```
void Mark3::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 153 of file ll.cpp.

19.3.3.3 operator new()

```
void* Mark3::CircularLinkList::operator new (
            size_t sz,
            void * pv ) [inline]
```

Definition at line 221 of file ll.h.

19.3.3.4 PivotBackward()

```
void Mark3::CircularLinkList::PivotBackward ( )
```

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

Definition at line 144 of file ll.cpp.

19.3.3.5 PivotForward()

```
void Mark3::CircularLinkList::PivotForward ( )
```

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

Definition at line 135 of file ll.cpp.

19.3.3.6 Remove()

```
void Mark3::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 103 of file ll.cpp.

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

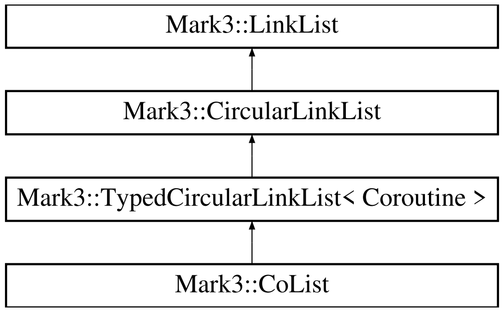
- /home/moslevin/projects/m3-repo/kernel/src/public/ll.h
- /home/moslevin/projects/m3-repo/kernel/src/ll.cpp

19.4 Mark3::CoList Class Reference

The CoList class The CoList class implements a circular-linked-listed structure for coroutine objects. The intent of this object is to maintain a list of active coroutine objects with a specific priority or state, to ensure that a freshly-schedulable co-routine always exists at the head of the list.

```
#include <colist.h>
```

Inheritance diagram for Mark3::CoList:



Public Member Functions

- void SetPrioMap (CoPrioMap ∗pclPrioMap_)

  SetPrioMap Assign a priority map object to this co-routine list.
- void SetPriority (PORT_PRIO_TYPE uPriority_)

  SetPriority Set the scheduling priority of this coroutine liss; has no effect unless a SetPrioMap has been called with a valid coroutine priority map object.
- void Add (Coroutine ∗pclCoroutine_)

  Add Add a coroutine object to this list.
- void Remove (Coroutine ∗pclCoroutine_)

  Remove Remove a given coroutine object from this list.

Private Attributes

- CoPrioMap ∗ m_pclPrioMap
- uint8_t m_uPriority

Additional Inherited Members

19.4.1 Detailed Description

The CoList class The CoList class implements a circular-linked-listed structure for coroutine objects. The intent of this object is to maintain a list of active coroutine objects with a specific priority or state, to ensure that a freshly-schedulable co-routine always exists at the head of the list.

Definition at line 35 of file colist.h.

19.4.2 Member Function Documentation

19.4.2.1 Add()

```
void Mark3::CoList::Add (
            Coroutine ∗ pclCoroutine_ )
```

Add Add a coroutine object to this list.

Parameters

*pcl↩ Coroutine_*	Pointer to the coroutine object to add

Definition at line 36 of file colist.cpp.

19.4.2.2 Remove()

```
void Mark3::CoList::Remove (
             Coroutine * pclCoroutine_ )
```

Remove Remove a given coroutine object from this list.

Parameters

*pcl↩ Coroutine_*	Pointer to the coroutine object to remove

Definition at line 46 of file colist.cpp.

19.4.2.3 SetPrioMap()

```
void Mark3::CoList::SetPrioMap (
             CoPrioMap * pclPrioMap_ )
```

SetPrioMap Assign a priority map object to this co-routine list.

Parameters

*pclPrio↩ Map_*	priority map object to assign

Definition at line 24 of file colist.cpp.

19.4.2.4 SetPriority()

```
void Mark3::CoList::SetPriority (
             PORT_PRIO_TYPE uPriority_ )
```

SetPriority Set the scheduling priority of this coroutine liss; has no effect unless a SetPrioMap has been called with a valid coroutine priority map object.

Parameters

u←↩ Priority←↩ _	Priority of coroutines associated with this list

Definition at line 30 of file colist.cpp.

19.4.3 Member Data Documentation

19.4.3.1 m_pclPrioMap

`CoPrioMap* Mark3::CoList::m_pclPrioMap [private]`

Definition at line 73 of file colist.h.

19.4.3.2 m_uPriority

`uint8_t Mark3::CoList::m_uPriority [private]`

Definition at line 74 of file colist.h.

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

- /home/moslevin/projects/m3-repo/kernel/src/public/colist.h
- /home/moslevin/projects/m3-repo/kernel/src/colist.cpp

19.5 Mark3::ConditionVariable Class Reference

The ConditionVariable class This class implements a condition variable. This is a synchronization object that allows multiple threads to block, each waiting for specific signals unique to them. Access to the specified condition is guarded by a mutex that is supplied by the caller. This object can permit multiple waiters that can be unblocked one-at-a-time via signalling, or unblocked all at once via broadcasting. This object is built upon lower-level primatives, and is somewhat more heavyweight than the primative types supplied by the kernel.

`#include <condvar.h>`

Public Member Functions

- void ∗ operator new (size_t sz, void ∗pv)
- void Init ()

  Init Initialize the condition variable prior to use. Must be called before the object can be used.
- void Wait (Mutex ∗pclMutex_)

  Wait Block the current thread, and wait for the object to be signalled. The specified mutex will be locked when the thread returns.
- bool Wait (Mutex ∗pclMutex_, uint32_t u32WaitTimeMS_)

  Wait Block the current thread, and wait for the object to be signalled. The specified mutex will be locked when the thread returns.
- void Signal ()

  Signal Signal/Unblock the next thread currently blocked on this condition variable.
- void Broadcast ()

  Broadcast Unblock all threads currently blocked on this condition variable.

Private Attributes

- Mutex m_clMutex
- Semaphore m_clSemaphore
- uint8_t m_u8Waiters

19.5.1 Detailed Description

The ConditionVariable class This class implements a condition variable. This is a synchronization object that allows multiple threads to block, each waiting for specific signals unique to them. Access to the specified condition is guarded by a mutex that is supplied by the caller. This object can permit multiple waiters that can be unblocked one-at-a-time via signalling, or unblocked all at once via broadcasting. This object is built upon lower-level primatives, and is somewhat more heavyweight than the primative types supplied by the kernel.

Definition at line 39 of file condvar.h.

19.5.2 Member Function Documentation

19.5.2.1 Broadcast()

```
void Mark3::ConditionVariable::Broadcast ( )
```

Broadcast Unblock all threads currently blocked on this condition variable.

Definition at line 77 of file condvar.cpp.

19.5.2.2 Init()

```
void Mark3::ConditionVariable::Init (
            void )
```

Init Initialize the condition variable prior to use. Must be called before the object can be used.

Definition at line 25 of file condvar.cpp.

19.5.2.3 operator new()

```
void* Mark3::ConditionVariable::operator new (
            size_t sz,
            void * pv ) [inline]
```

Definition at line 42 of file condvar.h.

19.5.2.4 Signal()

```
void Mark3::ConditionVariable::Signal ( )
```

Signal Signal/Unblock the next thread currently blocked on this condition variable.

Definition at line 66 of file condvar.cpp.

19.5.2.5 Wait() [1/2]

```
void Mark3::ConditionVariable::Wait (
            Mutex * pclMutex_ )
```

Wait Block the current thread, and wait for the object to be signalled. The specified mutex will be locked when the thread returns.

Parameters

*pcl↵ Mutex_*	Mutex to claim once the calling thread has access to the condvar

Definition at line 32 of file condvar.cpp.

19.5.2.6 Wait() [2/2]

```
bool Mark3::ConditionVariable::Wait (
            Mutex * pclMutex_,
            uint32_t u32WaitTimeMS_ )
```

Wait Block the current thread, and wait for the object to be signalled. The specified mutex will be locked when the thread returns.

Parameters

*pclMutex_*	Mutex to claim once the calling thread has access to the condvar
*u32WaitTimeMS_*	Maximum time in ms to wait before abandoning the operation

Returns

true on success, false on timeout

Definition at line 48 of file condvar.cpp.

19.5.3 Member Data Documentation

19.5.3.1 m_clMutex

```
Mutex Mark3::ConditionVariable::m_clMutex [private]
```

Definition at line 82 of file condvar.h.

19.5.3.2 m_clSemaphore

```
Semaphore Mark3::ConditionVariable::m_clSemaphore [private]
```

Definition at line 83 of file condvar.h.

19.5.3.3 m_u8Waiters

```
uint8_t Mark3::ConditionVariable::m_u8Waiters [private]
```

Definition at line 84 of file condvar.h.

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

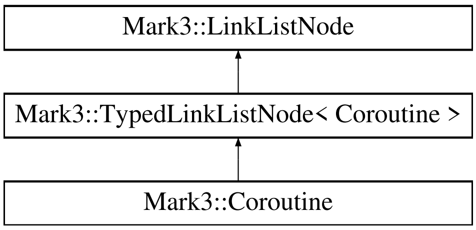
- /home/moslevin/projects/m3-repo/kernel/src/public/condvar.h
- /home/moslevin/projects/m3-repo/kernel/src/condvar.cpp

19.6 Mark3::Coroutine Class Reference

The Coroutine class implements a lightweight, run-to-completion task that forms the basis for co-operative task scheduling in Mark3. Coroutines are designed to be run from a singular context, and scheduled as a result of events occurring from threads, timers, interrupt sources, or other co-routines.

```
#include <coroutine.h>
```

Inheritance diagram for Mark3::Coroutine:



Public Member Functions

- ∼Coroutine ()
- void Init (PORT_PRIO_TYPE uPriority_, CoroutineHandler pfHandler_, void ∗pvContext_)

  Init Initialize the coroutine object prior to use. Must be called before using the other methods in the class.
- void Run ()

  Run Clear the co-routine's pending execution flag, and execute the coroutine's handler function.
- void Activate ()

  Activate Tag the co-routine as pending execution. Has no effect if the co-routine is already pending execution.
- void SetPriority (PORT_PRIO_TYPE uPriority_)

  SetPriority Update the scheduling priority of the co-routine. Can be called from within the co-routine, or from any other context aware of the co-routine object.
- PORT_PRIO_TYPE GetPriority ()

  GetPriority Retrieve the current scheduling priority of the co-routine.

Private Attributes

- CoList ∗ m_pclOwner
- CoroutineHandler m_pfHandler
- void ∗ m_pvContext
- PORT_PRIO_TYPE m_uPriority
- bool m_bQueued

Additional Inherited Members

19.6.1 Detailed Description

The Coroutine class implements a lightweight, run-to-completion task that forms the basis for co-operative task scheduling in Mark3. Coroutines are designed to be run from a singular context, and scheduled as a result of events occurring from threads, timers, interrupt sources, or other co-routines.

Co-routines differ from Threads in that they cannot block, and must run to completion before other (potentially higher-priority) co-routines block.

Examples:

lab2_coroutines/main.cpp.

Definition at line 53 of file coroutine.h.

19.6.2 Constructor & Destructor Documentation

19.6.2.1 ∼Coroutine()

```
Mark3::Coroutine::∼Coroutine ( )
```

Definition at line 28 of file coroutine.cpp.

19.6.3 Member Function Documentation

19.6.3.1 Activate()

```
void Mark3::Coroutine::Activate ( )
```

Activate Tag the co-routine as pending execution. Has no effect if the co-routine is already pending execution.

Examples:

lab2_coroutines/main.cpp.

Definition at line 67 of file coroutine.cpp.

19.6.3.2 GetPriority()

```
PORT_PRIO_TYPE Mark3::Coroutine::GetPriority ( )
```

GetPriority Retrieve the current scheduling priority of the co-routine.

Returns

current scheduling priority of the co-routine.

Definition at line 97 of file coroutine.cpp.

19.6.3.3 Init()

```
void Mark3::Coroutine::Init (
            PORT_PRIO_TYPE uPriority_,
            CoroutineHandler pfHandler_,
            void * pvContext_ )
```

Init Initialize the coroutine object prior to use. Must be called before using the other methods in the class.

Parameters

*uPriority_*	The scheduling priority of this coroutine as configured by the port.
*pfHandler_*	Function to run when the coroutine is executed
*pvContext_*	User-defined value passed into the handler function on execution

Definition at line 39 of file coroutine.cpp.

19.6.3.4 Run()

```
void Mark3::Coroutine::Run ( )
```

Run Clear the co-routine's pending execution flag, and execute the coroutine's handler function.

Definition at line 53 of file coroutine.cpp.

19.6.3.5 SetPriority()

```
void Mark3::Coroutine::SetPriority (
            PORT_PRIO_TYPE uPriority_ )
```

SetPriority Update the scheduling priority of the co-routine. Can be called from within the co-routine, or from any other context aware of the co-routine object.

Parameters

*uPriority_*	New priority of the co-routine

Definition at line 82 of file coroutine.cpp.

19.6.4 Member Data Documentation

19.6.4.1 m_bQueued

```
bool Mark3::Coroutine::m_bQueued [private]
```

Definition at line 108 of file coroutine.h.

19.6.4.2 m_pclOwner

```
CoList* Mark3::Coroutine::m_pclOwner [private]
```

Definition at line 104 of file coroutine.h.

19.6.4.3 m_pfHandler

```
CoroutineHandler Mark3::Coroutine::m_pfHandler [private]
```

Definition at line 105 of file coroutine.h.

19.6.4.4 m_pvContext

```
void* Mark3::Coroutine::m_pvContext [private]
```

Definition at line 106 of file coroutine.h.

19.6.4.5 m_uPriority

```
PORT_PRIO_TYPE Mark3::Coroutine::m_uPriority [private]
```

Definition at line 107 of file coroutine.h.

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

- /home/moslevin/projects/m3-repo/kernel/src/public/coroutine.h
- /home/moslevin/projects/m3-repo/kernel/src/coroutine.cpp

19.7 Mark3::CoScheduler Class Reference

The CoScheduler class. This class implements the coroutine scheduler. Similar to the Mark3 thread scheduler, the highest-priority active object is scheduled / returned for execution. If no active co-routines are available to be scheduled, then the scheduler returns nullptr.

```
#include <cosched.h>
```

Static Public Member Functions

- static void Init ()

  Init Initialize the coroutine scheduler prior to use. Must be called prior to using any other functions in the coroutine scheduler.
- static CoPrioMap ∗ GetPrioMap ()

  GetPrioMap Get the pointer to the priority map object used by the scheduler.
- static CoList ∗ GetStopList ()

  GetStopList Get the pointer to the coroutine list managing initialized coroutines that are not awaiting execution.
- static CoList ∗ GetCoList (PORT_PRIO_TYPE uPriority_)

  GetCoList Retrieve the coroutine list associated with a given priority.
- static Coroutine ∗ Schedule ()

  Schedule Determine what coroutine (if any) is the next to be executed.

Static Private Attributes

- static CoList m_aclPriorities [PORT_COROUTINE_PRIORITIES]
- static CoList m_clStopList
- static CoPrioMap m_clPrioMap

19.7.1 Detailed Description

The CoScheduler class. This class implements the coroutine scheduler. Similar to the Mark3 thread scheduler, the highest-priority active object is scheduled / returned for execution. If no active co-routines are available to be scheduled, then the scheduler returns nullptr.

Definition at line 33 of file cosched.h.

19.7.2 Member Function Documentation

19.7.2.1 GetCoList()

```
CoList * Mark3::CoScheduler::GetCoList (
            PORT_PRIO_TYPE uPriority_ )  [static]
```

GetCoList Retrieve the coroutine list associated with a given priority.

Parameters

*uPriority_*	Priority to get the coroutine list of.

Returns

coroutine list pointer or nullptr on invalid priority.

Definition at line 51 of file cosched.cpp.

19.7.2.2 GetPrioMap()

```
CoPrioMap * Mark3::CoScheduler::GetPrioMap ( ) [static]
```

GetPrioMap Get the pointer to the priority map object used by the scheduler.

Returns

Return the priority map object owned by the schedule

Definition at line 39 of file cosched.cpp.

19.7.2.3 GetStopList()

```
CoList * Mark3::CoScheduler::GetStopList ( ) [static]
```

GetStopList Get the pointer to the coroutine list managing initialized coroutines that are not awaiting execution.

Returns

Pointer to the coroutine stop list

Definition at line 45 of file cosched.cpp.

19.7.2.4 Init()

```
void Mark3::CoScheduler::Init (
             void ) [static]
```

Init Initialize the coroutine scheduler prior to use. Must be called prior to using any other functions in the coroutine scheduler.

Examples:

lab2_coroutines/main.cpp.

Definition at line 29 of file cosched.cpp.

19.7.2.5 Schedule()

`Coroutine * Mark3::CoScheduler::Schedule ( ) [static]`

Schedule Determine what coroutine (if any) is the next to be executed.

Returns

next coroutine to execute, or nullptr if no coroutines are ready to be scheduled.

Examples:

lab2_coroutines/main.cpp.

Definition at line 60 of file cosched.cpp.

19.7.3 Member Data Documentation

19.7.3.1 m_aclPriorities

`CoList Mark3::CoScheduler::m_aclPriorities [static], [private]`

Definition at line 80 of file cosched.h.

19.7.3.2 m_clPrioMap

`CoPrioMap Mark3::CoScheduler::m_clPrioMap [static], [private]`

Definition at line 82 of file cosched.h.

19.7.3.3 m_clStopList

`CoList Mark3::CoScheduler::m_clStopList [static], [private]`

Definition at line 81 of file cosched.h.

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

- /home/moslevin/projects/m3-repo/kernel/src/public/cosched.h
- /home/moslevin/projects/m3-repo/kernel/src/cosched.cpp

19.8 Mark3::CriticalGuard Class Reference

The CriticalGuard class. This class provides an implemention of RAII for critical sections. Object creation results in a critical section being invoked. The subsequent destructor call results in the critical section being released.

```
#include <criticalguard.h>
```

Public Member Functions

- CriticalGuard ()
- ∼CriticalGuard ()

19.8.1 Detailed Description

The CriticalGuard class. This class provides an implemention of RAII for critical sections. Object creation results in a critical section being invoked. The subsequent destructor call results in the critical section being released.

This is similar to the LockGuard class, except that class operates on a single Mutex, and this class operates on a global interrupt-disabled lock.

Definition at line 38 of file criticalguard.h.

19.8.2 Constructor & Destructor Documentation

19.8.2.1 CriticalGuard()

```
Mark3::CriticalGuard::CriticalGuard ( )  [inline]
```

Definition at line 40 of file criticalguard.h.

19.8.2.2 ∼CriticalGuard()

```
Mark3::CriticalGuard::~CriticalGuard ( )  [inline]
```

Definition at line 44 of file criticalguard.h.

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

- /home/moslevin/projects/m3-repo/kernel/src/public/criticalguard.h

19.9 Mark3::CriticalSection Class Reference

The CriticalSection class. This class implements a portable CriticalSection interface based on macros/inline functions that are implemented as part of each port.

`#include <criticalsection.h>`

Static Public Member Functions

- static void Enter ()

  Enter Enter a critical section, disabling all kernel-aware interrupts, and giving exclusive control of the CPU to the curreninttly running task.
- static void Exit ()

  Exit Exit a critical section, re-enabling kernel-aware interrupts, and releasing exclusive control of the CPU.
- static K_WORD NestingCount ()

  NestingCount.

19.9.1 Detailed Description

The CriticalSection class. This class implements a portable CriticalSection interface based on macros/inline functions that are implemented as part of each port.

Critical sections *can* be safely nested, that is, multiple calls to Enter() may be called before matching calls to Exit() are called. In such cases, only the *final* call to Exit() will cause the caller to relinquish control of the CPU.

Critical sections can be safely used within interrupts; although they have no effect in the general case. The exception is in the case where a system has multiple levels of interrupt nesting; at which point calling a critical section from an interrupt temporarily disables subsequent levels of nesting.

Care, however, must be taken to ensure that the currently executing thread does *not* block when a critical section is active. This condition is guaranteed to break the system.

Definition at line 48 of file criticalsection.h.

19.9.2 Member Function Documentation

19.9.2.1 Enter()

```
static void Mark3::CriticalSection::Enter ( ) [inline], [static]
```

Enter Enter a critical section, disabling all kernel-aware interrupts, and giving exclusive control of the CPU to the curreninttly running task.

Examples:

lab9_dynamic_threads/main.cpp.

Definition at line 56 of file criticalsection.h.

19.9.2.2 Exit()

```
static void Mark3::CriticalSection::Exit ( )  [inline], [static]
```

Exit Exit a critical section, re-enabling kernel-aware interrupts, and releasing exclusive control of the CPU.

Examples:

lab9_dynamic_threads/main.cpp.

Definition at line 65 of file criticalsection.h.

19.9.2.3 NestingCount()

```
static K_WORD Mark3::CriticalSection::NestingCount ( )  [inline], [static]
```

NestingCount.

Returns

Number of Enter() calls awaiting an Exit() call.

Definition at line 73 of file criticalsection.h.

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

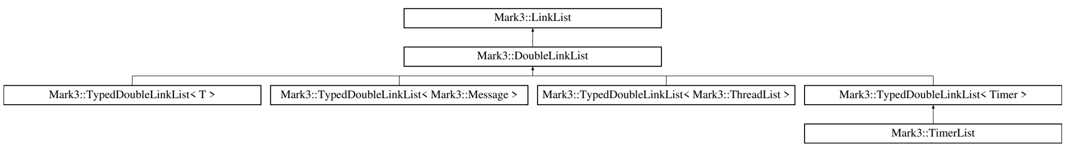
- /home/moslevin/projects/m3-repo/kernel/src/public/criticalsection.h

19.10 Mark3::DoubleLinkList Class Reference

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

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

Inheritance diagram for Mark3::DoubleLinkList:



Public Member Functions

- void ∗ operator new (size_t sz, void ∗pv)
- DoubleLinkList ()

  DoubleLinkList.
- void Add (LinkListNode ∗node_)

  Add.
- void Remove (LinkListNode ∗node_)

  Remove.

Additional Inherited Members

19.10.1 Detailed Description

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

Definition at line 179 of file ll.h.

19.10.2 Constructor & Destructor Documentation

19.10.2.1 DoubleLinkList()

```
Mark3::DoubleLinkList::DoubleLinkList ( )  [inline]
```

DoubleLinkList.

Default constructor - initializes the head/tail nodes to nullptr

Definition at line 188 of file ll.h.

19.10.3 Member Function Documentation

19.10.3.1 Add()

```
void Mark3::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 34 of file ll.cpp.

19.10.3.2 operator new()

```
void* Mark3::DoubleLinkList::operator new (
```

```
            size_t sz,
            void * pv )  [inline]
```

Definition at line 182 of file ll.h.

19.10.3.3 Remove()

```
void Mark3::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 55 of file ll.cpp.

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

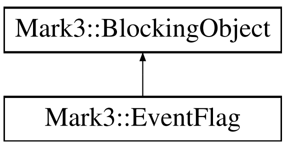
- /home/moslevin/projects/m3-repo/kernel/src/public/ll.h
- /home/moslevin/projects/m3-repo/kernel/src/ll.cpp

19.11 Mark3::EventFlag Class Reference

The EventFlag class. This class implements 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.

```
#include <eventflag.h>
```

Inheritance diagram for Mark3::EventFlag:



Public Member Functions

- void ∗ operator new (size_t sz, void ∗pv)
- ∼EventFlag ()
- void Init ()

    Init Initializes the EventFlag object prior to use.
- uint16_t Wait (uint16_t u16Mask_, EventFlagOperation eMode_)

    Wait Block a thread on the specific flags in this event flag group.
- uint16_t Wait (uint16_t u16Mask_, EventFlagOperation eMode_, uint32_t u32TimeMS_)

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

    WakeMe Wake the given thread, currently blocking on this object.
- void 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.
- 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 eMode_, uint32_t u32TimeMS_)

    Wait_i Interal abstraction used to manage both timed and untimed wait operations.

Private Attributes

- uint16_t m_u16SetMask

    Event flags currently set in this object.

Additional Inherited Members

19.11.1 Detailed Description

The EventFlag class. This class implements 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 45 of file eventflag.h.

19.11.2 Constructor & Destructor Documentation

19.11.2.1 ∼EventFlag()

```
Mark3::EventFlag::∼EventFlag ( )
```

19.11.3 Member Function Documentation

19.11.3.1 Clear()

```
void Mark3::EventFlag::Clear (
            uint16_t u16Mask_ )
```

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

Parameters

u16↩ *Mask_*	- Bitmask of flags to clear

Examples:

lab7_events/main.cpp.

19.11.3.2 GetMask()

```
uint16_t Mark3::EventFlag::GetMask ( )
```

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

Returns

The state of the 16-bit bitmask

19.11.3.3 Init()

```
void Mark3::EventFlag::Init ( )
```

Init Initializes the EventFlag object prior to use.

19.11.3.4 operator new()

```
void* Mark3::EventFlag::operator new (
            size_t sz,
            void * pv )  [inline]
```

Definition at line 48 of file eventflag.h.

19.11.3.5 Set()

```
void Mark3::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.

19.11.3.6 Wait() [1/2]

```
uint16_t Mark3::EventFlag::Wait (
            uint16_t u16Mask_,
            EventFlagOperation eMode_ )
```

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

Parameters

*u16Mask_*	- 16-bit bitmask to block on
*eMode_*	- EventFlagOperation::Any_Set: Thread will block on any of the bits in the mask • EventFlagOperation::All_Set: 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.

19.11.3.7 Wait() [2/2]

```
uint16_t Mark3::EventFlag::Wait (
              uint16_t u16Mask_,
              EventFlagOperation 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_*	- EventFlagOperation::Any_Set: Thread will block on any of the bits in the mask • EventFlagOperation::All_Set: Thread will block on all of the bits in the mask
*u32TimeM↩ S_*	- Time to block (in ms)

Returns

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

19.11.3.8 Wait_i()

```
uint16_t Mark3::EventFlag::Wait_i (
              uint16_t u16Mask_,
              EventFlagOperation 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_*	- EventFlagOperation::Any_Set: Thread will block on any of the bits in the mask • EventFlagOperation::All_Set: Thread will block on all of the bits in the mask
*u32TimeM↩ S_*	- Time to block (in ms)

Returns

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

19.11.3.9 WakeMe()

```
void Mark3::EventFlag::WakeMe (
            Thread * pclChosenOne_ )
```

WakeMe Wake the given thread, currently blocking on this object.

Parameters

*pclChosen↩ One_*	Pointer to the owner thread to unblock.

19.11.4 Member Data Documentation

19.11.4.1 m_u16SetMask

```
uint16_t Mark3::EventFlag::m_u16SetMask  [private]
```

Event flags currently set in this object.

Definition at line 119 of file eventflag.h.

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

- /home/moslevin/projects/m3-repo/kernel/src/public/eventflag.h

19.12 Mark3::Kernel Class Reference

The Kernel Class encapsulates all of the kernel startup, configuration and management functions.

```
#include <kernel.h>
```

Static Public Member Functions

- static void Init ()

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

  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 void CompleteStart ()

  CompleteStart Call this from the thread initialization code at the point that the scheduler is to be run for the first time.
- static bool IsStarted ()

  IsStarted.
- static void SetPanic (PanicFunc 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 SetThreadCreateCallout (ThreadCreateCallout pfCreate_)

  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.
- static void SetThreadExitCallout (ThreadExitCallout pfExit_)

  SetThreadExitCallout Set a function to be called on thread exit. This callout is executed from the beginning of Thread::Exit().
- static void SetThreadContextSwitchCallout (ThreadContextCallout pfContext_)

  SetThreadContextSwitchCallout Set a function to be called on each context switch.
- static void SetDebugPrintFunction (DebugPrintFunction pfPrintFunction_)

  SetDebugPrintFunction Set the function to be used when printing kernel debug information.
- static void DebugPrint (const char ∗szString_)

  DebugPrint Print a string to the configured output interface. Has no effect if Kernel::SetDebugPrintFunction() has not been called with a valid print handler.
- static ThreadCreateCallout GetThreadCreateCallout ()

  GetThreadCreateCallout Return the current function called on every Thread::Init();.
- static ThreadExitCallout GetThreadExitCallout ()

  GetThreadExitCallout Return the current function called on every Thread::Exit();.
- static ThreadContextCallout GetThreadContextSwitchCallout ()

  GetThreadContextSwitchCallout Return the current function called on every Thread::ContextSwitchSWI()
- static void SetStackGuardThreshold (uint16_t u16Threshold_)
- static uint16_t GetStackGuardThreshold ()
- static void Tick ()
- static uint32_t GetTicks ()

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 m_pfPanic

  set panic function
- static ThreadCreateCallout m_pfThreadCreateCallout

Function to call on thread creation.
- static ThreadExitCallout m_pfThreadExitCallout

  Function to call on thread exit.
- static ThreadContextCallout m_pfThreadContextCallout

  Function to call on context switch.
- static DebugPrintFunction m_pfDebugPrintFunction

  Function to call to print debug info.
- static uint16_t m_u16GuardThreshold
- static uint32_t m_u32Ticks

19.12.1 Detailed Description

The Kernel Class encapsulates all of the kernel startup, configuration and management functions.

Definition at line 48 of file kernel.h.

19.12.2 Member Function Documentation

19.12.2.1 CompleteStart()

```
void Mark3::Kernel::CompleteStart ( )  [static]
```

CompleteStart Call this from the thread initialization code at the point that the scheduler is to be run for the first time.

Definition at line 64 of file kernel.cpp.

19.12.2.2 DebugPrint()

```
void Mark3::Kernel::DebugPrint (
            const char * szString_ )  [static]
```

DebugPrint Print a string to the configured output interface. Has no effect if Kernel::SetDebugPrintFunction() has not been called with a valid print handler.

Parameters

*sz←String_*	string to print

Examples:

lab10_notifications/main.cpp, lab11_mailboxes/main.cpp, lab1_kernel_setup/main.cpp, lab2_coroutines/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 81 of file kernel.cpp.

19.12.2.3 GetStackGuardThreshold()

```
static uint16_t Mark3::Kernel::GetStackGuardThreshold ( )  [inline], [static]
```

Definition at line 202 of file kernel.h.

19.12.2.4 GetThreadContextSwitchCallout()

```
static ThreadContextCallout Mark3::Kernel::GetThreadContextSwitchCallout ( )  [inline], [static]
```

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

Returns

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

Definition at line 198 of file kernel.h.

19.12.2.5 GetThreadCreateCallout()

```
static ThreadCreateCallout Mark3::Kernel::GetThreadCreateCallout ( )  [inline], [static]
```

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

Returns

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

Definition at line 178 of file kernel.h.

19.12.2.6 GetThreadExitCallout()

```
static ThreadExitCallout Mark3::Kernel::GetThreadExitCallout ( )  [inline], [static]
```

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

Returns

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

Definition at line 188 of file kernel.h.

19.12.2.7 GetTicks()

```
uint32_t Mark3::Kernel::GetTicks ( )  [static]
```

Examples:

lab9_dynamic_threads/main.cpp.

Definition at line 90 of file kernel.cpp.

19.12.2.8 Init()

```
void Mark3::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:

lab10_notifications/main.cpp, lab11_mailboxes/main.cpp, lab1_kernel_setup/main.cpp, lab2_coroutines/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 44 of file kernel.cpp.

19.12.2.9 IsPanic()

```
static bool Mark3::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 99 of file kernel.h.

19.12.2.10 IsStarted()

```
static bool Mark3::Kernel::IsStarted ( )  [inline], [static]
```

IsStarted.

Returns

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

Definition at line 86 of file kernel.h.

19.12.2.11 Panic()

```
void Mark3::Kernel::Panic (
            uint16_t u16Cause_ )  [static]
```

Panic Cause the kernel to enter its panic state.

Parameters

*u16←↩ Cause_*	Reason for the kernel panic

Definition at line 70 of file kernel.cpp.

19.12.2.12 SetDebugPrintFunction()

```
static void Mark3::Kernel::SetDebugPrintFunction (
            DebugPrintFunction pfPrintFunction_ )  [inline], [static]
```

SetDebugPrintFunction Set the function to be used when printing kernel debug information.

Parameters

*pfPrint←↩ Function_*	Function used to print kernel debug message strings

Examples:

lab10_notifications/main.cpp, lab11_mailboxes/main.cpp, lab1_kernel_setup/main.cpp, lab2_coroutines/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 155 of file kernel.h.

19.12.2.13 SetPanic()

```
static void Mark3::Kernel::SetPanic (
            PanicFunc pfPanic_ )  [inline], [static]
```

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

Parameters

pf←↩ Panic←↩ _	Panic function pointer

Definition at line 94 of file kernel.h.

19.12.2.14 SetStackGuardThreshold()

```
static void Mark3::Kernel::SetStackGuardThreshold (
             uint16_t u16Threshold_ )  [inline], [static]
```

Definition at line 201 of file kernel.h.

19.12.2.15 SetThreadContextSwitchCallout()

```
static void Mark3::Kernel::SetThreadContextSwitchCallout (
             ThreadContextCallout 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

pf↵ Context↵ _	Pointer to a function to call on context switch

Examples:

lab9_dynamic_threads/main.cpp.

Definition at line 143 of file kernel.h.

19.12.2.16 SetThreadCreateCallout()

```
static void Mark3::Kernel::SetThreadCreateCallout (
             ThreadCreateCallout 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

pf↵ Create↵ _	Pointer to a function to call on thread creation

Examples:

lab9_dynamic_threads/main.cpp.

Definition at line 116 of file kernel.h.

19.12.2.17 SetThreadExitCallout()

```
static void Mark3::Kernel::SetThreadExitCallout (
            ThreadExitCallout 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

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

Examples:

lab9_dynamic_threads/main.cpp.

Definition at line 130 of file kernel.h.

19.12.2.18 Start()

```
void Mark3::Kernel::Start ( )  [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:

lab10_notifications/main.cpp, lab11_mailboxes/main.cpp, lab1_kernel_setup/main.cpp, lab2_coroutines/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 58 of file kernel.cpp.

19.12.2.19 Tick()

`static void Mark3::Kernel::Tick ( ) [inline], [static]`

Definition at line 205 of file kernel.h.

19.12.3 Member Data Documentation

19.12.3.1 m_bIsPanic

`bool Mark3::Kernel::m_bIsPanic [static], [private]`

true if kernel is in panic state, false otherwise

Definition at line 210 of file kernel.h.

19.12.3.2 m_bIsStarted

`bool Mark3::Kernel::m_bIsStarted [static], [private]`

true if kernel is running, false otherwise

Definition at line 209 of file kernel.h.

19.12.3.3 m_pfDebugPrintFunction

`DebugPrintFunction Mark3::Kernel::m_pfDebugPrintFunction [static], [private]`

Function to call to print debug info.

Function to call when printing debug info.

Definition at line 222 of file kernel.h.

19.12.3.4 m_pfPanic

`PanicFunc Mark3::Kernel::m_pfPanic [static], [private]`

set panic function

Definition at line 211 of file kernel.h.

19.12.3.5 m_pfThreadContextCallout

`ThreadContextCallout Mark3::Kernel::m_pfThreadContextCallout [static], [private]`

Function to call on context switch.

Definition at line 220 of file kernel.h.

19.12.3.6 m_pfThreadCreateCallout

`ThreadCreateCallout Mark3::Kernel::m_pfThreadCreateCallout [static], [private]`

Function to call on thread creation.

Definition at line 214 of file kernel.h.

19.12.3.7 m_pfThreadExitCallout

`ThreadExitCallout Mark3::Kernel::m_pfThreadExitCallout [static], [private]`

Function to call on thread exit.

Definition at line 217 of file kernel.h.

19.12.3.8 m_u16GuardThreshold

`uint16_t Mark3::Kernel::m_u16GuardThreshold [static], [private]`

Definition at line 224 of file kernel.h.

19.12.3.9 m_u32Ticks

`uint32_t Mark3::Kernel::m_u32Ticks [static], [private]`

Definition at line 226 of file kernel.h.

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

- /home/moslevin/projects/m3-repo/kernel/src/public/kernel.h
- /home/moslevin/projects/m3-repo/kernel/src/kernel.cpp

19.13 Mark3::KernelSWI Class Reference

The KernelSWI Class provides the software-interrupt used to implement the context-switching interrupt used by the kernel. This interface must be implemented by target-specific code in the porting layer.

```
#include <kernelswi.h>
```

Static Public Member Functions

- static void Config (void)

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

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

  Trigger Call the software interrupt.

19.13.1 Detailed Description

The KernelSWI Class provides the software-interrupt used to implement the context-switching interrupt used by the kernel. This interface must be implemented by target-specific code in the porting layer.

Definition at line 32 of file kernelswi.h.

19.13.2 Member Function Documentation

19.13.2.1 Config()

```
void Mark3::KernelSWI::Config (
            void ) [static]
```

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

Definition at line 31 of file kernelswi.cpp.

19.13.2.2 Start()

```
void Mark3::KernelSWI::Start (
            void ) [static]
```

Start Enable ("Start") the software interrupt functionality.

Definition at line 39 of file kernelswi.cpp.

19.13.2.3 Trigger()

```
void Mark3::KernelSWI::Trigger (
            void ) [static]
```

Trigger Call the software interrupt.

Definition at line 46 of file kernelswi.cpp.

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

- /home/moslevin/projects/m3-repo/kernel/src/public/kernelswi.h
- /home/moslevin/projects/m3-repo/kernel/src/arch/avr/atmega1284p/gcc/kernelswi.cpp

19.14 Mark3::KernelTimer Class Reference

The KernelTimer class provides a timer interface used by all time-based scheduling/timer subsystems in the kernel. This interface must be implemented by target-specific code in the porting layer.

```
#include <kerneltimer.h>
```

Static Public Member Functions

- static void Config (void)

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

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

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

19.14.1 Detailed Description

The KernelTimer class provides a timer interface used by all time-based scheduling/timer subsystems in the kernel. This interface must be implemented by target-specific code in the porting layer.

Definition at line 34 of file kerneltimer.h.

19.14.2 Member Function Documentation

19.14.2.1 Config()

```
void Mark3::KernelTimer::Config (
            void ) [static]
```

Config Initializes the kernel timer before use.

Definition at line 60 of file kerneltimer.cpp.

19.14.2.2 Start()

```
void Mark3::KernelTimer::Start (
            void ) [static]
```

Start Starts the kernel time (must be configured first)

Definition at line 76 of file kerneltimer.cpp.

19.14.2.3 Stop()

```
void Mark3::KernelTimer::Stop (
            void ) [static]
```

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

Definition at line 86 of file kerneltimer.cpp.

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

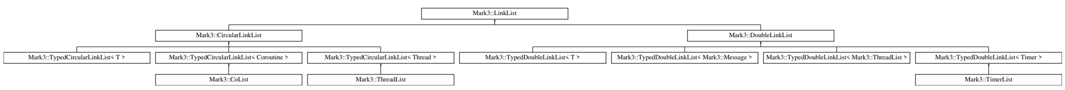
- /home/moslevin/projects/m3-repo/kernel/src/public/kerneltimer.h
- /home/moslevin/projects/m3-repo/kernel/src/arch/avr/atmega1284p/gcc/kerneltimer.cpp

19.15 Mark3::LinkList Class Reference

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

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

Inheritance diagram for Mark3::LinkList:



Public Member Functions

- void Init ()
  Init.
- LinkListNode ∗ GetHead ()
  GetHead.
- void SetHead (LinkListNode ∗pclNode_)
  SetHead.
- LinkListNode ∗ GetTail ()
  GetTail.
- void SetTail (LinkListNode ∗pclNode_)
  SetTail.

Protected Attributes

- LinkListNode ∗ m_pclHead

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

    Pointer to the tail node in the list.

19.15.1 Detailed Description

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

Definition at line 119 of file ll.h.

19.15.2 Member Function Documentation

19.15.2.1 GetHead()

```
LinkListNode* Mark3::LinkList::GetHead ( )  [inline]
```

GetHead.

Get the head node in the linked list

Returns

Pointer to the head node in the list

Definition at line 144 of file ll.h.

19.15.2.2 GetTail()

```
LinkListNode* Mark3::LinkList::GetTail ( )  [inline]
```

GetTail.

Get the tail node of the linked list

Returns

Pointer to the tail node in the list

Definition at line 162 of file ll.h.

19.15.2.3 Init()

```
void Mark3::LinkList::Init (
            void ) [inline]
```

Init.

Clear the linked list.

Definition at line 131 of file ll.h.

19.15.2.4 SetHead()

```
void Mark3::LinkList::SetHead (
            LinkListNode * pclNode_ ) [inline]
```

SetHead.

Set the head node of a linked list

Parameters

*pcl↵ Node_*	Pointer to node to set as the head of the linked list

Definition at line 153 of file ll.h.

19.15.2.5 SetTail()

```
void Mark3::LinkList::SetTail (
            LinkListNode * pclNode_ ) [inline]
```

SetTail.

Set the tail node of the linked list

Parameters

*pcl↵ Node_*	Pointer to the node to set as the tail of the linked list

Definition at line 171 of file ll.h.

19.15.3 Member Data Documentation

19.15.3.1 m_pclHead

`LinkListNode* Mark3::LinkList::m_pclHead [protected]`

Pointer to the head node in the list.

Definition at line 122 of file ll.h.

19.15.3.2 m_pclTail

`LinkListNode* Mark3::LinkList::m_pclTail [protected]`

Pointer to the tail node in the list.

Definition at line 123 of file ll.h.

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

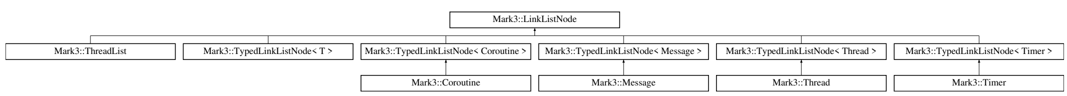
- /home/moslevin/projects/m3-repo/kernel/src/public/ll.h

19.16 Mark3::LinkListNode Class Reference

The LinkListNode Class Basic linked-list node data structure. This data is managed by the linked-list class types, and can be used transparently between them.

`#include <ll.h>`

Inheritance diagram for Mark3::LinkListNode:



Public Member Functions

- LinkListNode ∗ GetNext (void)

  GetNext.
- LinkListNode ∗ GetPrev (void)

  GetPrev.

Protected Member Functions

- LinkListNode ()
- 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

19.16.1 Detailed Description

The LinkListNode Class 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 62 of file ll.h.

19.16.2 Constructor & Destructor Documentation

19.16.2.1 LinkListNode()

```
Mark3::LinkListNode::LinkListNode ( )  [inline], [protected]
```

Definition at line 68 of file ll.h.

19.16.3 Member Function Documentation

19.16.3.1 ClearNode()

```
void Mark3::LinkListNode::ClearNode ( )  [protected]
```

ClearNode.

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

Definition at line 27 of file ll.cpp.

19.16.3.2 GetNext()

```
LinkListNode* Mark3::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 85 of file ll.h.

19.16.3.3 GetPrev()

```
LinkListNode* Mark3::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 93 of file ll.h.

19.16.4 Friends And Related Function Documentation

19.16.4.1 CircularLinkList

```
friend class CircularLinkList [friend]
```

Definition at line 96 of file ll.h.

19.16.4.2 DoubleLinkList

```
friend class DoubleLinkList [friend]
```

Definition at line 95 of file ll.h.

19.16.4.3 LinkList

```
friend class LinkList [friend]
```

Definition at line 94 of file ll.h.

19.16.5 Member Data Documentation

19.16.5.1 next

```
LinkListNode* Mark3::LinkListNode::next [protected]
```

Pointer to the next node in the list.

Definition at line 65 of file ll.h.

19.16.5.2 prev

```
LinkListNode* Mark3::LinkListNode::prev [protected]
```

Pointer to the previous node in the list.

Definition at line 66 of file ll.h.

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

- /home/moslevin/projects/m3-repo/kernel/src/public/ll.h
- /home/moslevin/projects/m3-repo/kernel/src/ll.cpp

19.17 Mark3::LockGuard Class Reference

The LockGuard class. This class provides RAII locks based on Mark3's kernel Mutex object. Note that Mark3 does not support exceptions, so care must be taken to ensure that this object is only used where that constraint can be met.

```
#include <lockguard.h>
```

Public Member Functions

- LockGuard (Mutex *pclMutex)
- LockGuard (Mutex *pclMutex, uint32_t u32TimeoutMs_)
- ~LockGuard ()
- bool isAcquired ()

Private Attributes

- bool m_bIsAcquired
- Mutex * m_pclMutex

19.17.1 Detailed Description

The LockGuard class. This class provides RAII locks based on Mark3's kernel Mutex object. Note that Mark3 does not support exceptions, so care must be taken to ensure that this object is only used where that constraint can be met.

Definition at line 32 of file lockguard.h.

19.17.2 Constructor & Destructor Documentation

19.17.2.1 LockGuard() [1/2]

```
Mark3::LockGuard::LockGuard (
            Mutex * pclMutex )
```

Parameters

pclMutex	mutex to lock during construction

Definition at line 25 of file lockguard.cpp.

19.17.2.2 LockGuard() [2/2]

```
Mark3::LockGuard::LockGuard (
            Mutex * pclMutex,
            uint32_t u32TimeoutMs_ )
```

Parameters

pclMutex	mutex to lock during construction
*u32Timeout↩ Ms_*	timeout (in ms) to wait before bailng

Definition at line 34 of file lockguard.cpp.

19.17.2.3 ∼LockGuard()

```
Mark3::LockGuard::∼LockGuard ( )
```

Definition at line 42 of file lockguard.cpp.

19.17.3 Member Function Documentation

19.17.3.1 isAcquired()

```
bool Mark3::LockGuard::isAcquired ( ) [inline]
```

Verify that lock was correctly initialized and locked during acquisition. This is used to provide error-checking for timed RAII locks, where Mark3 does not use exceptions, and a kernel-panic is too heavy-handed.

Returns

true if the lock was initialed correctly, false on error

Definition at line 54 of file lockguard.h.

19.17.4 Member Data Documentation

19.17.4.1 m_bIsAcquired

```
bool Mark3::LockGuard::m_bIsAcquired [private]
```

Definition at line 57 of file lockguard.h.

19.17.4.2 m_pclMutex

```
Mutex* Mark3::LockGuard::m_pclMutex [private]
```

Definition at line 58 of file lockguard.h.

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

- /home/moslevin/projects/m3-repo/kernel/src/public/lockguard.h
- /home/moslevin/projects/m3-repo/kernel/src/lockguard.cpp

19.18 Mark3::Mailbox Class Reference

The Mailbox class. This class implements an IPC mechnism based on sending/receiving 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 ∗ operator new (size_t sz, void ∗pv)
- ∼Mailbox ()
- void 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.
- bool 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.
- bool 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.
- bool 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.
- bool 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.
- void 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.
- void 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.
- bool 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.
- bool 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.
- uint16_t GetFreeSlots (void)
- bool IsFull (void)
- bool IsEmpty (void)

Static Public Member Functions

- static Mailbox ∗ Init (uint16_t u16BufferSize_, uint16_t u16ElementSize_)

  Init Create and initialize the mailbox object prior to its use. This must be called before any calls can be made to the object. This version of the API alloctes the buffer space from the kernel's Auto-Allocation heap, which cannot be returned back. As a result, this is only suitable for cases where the mailbox will be created once on startup, and persist for the duration of the system.

Private Member Functions

- void ∗ GetHeadPointer (void)

  GetHeadPointer Return a pointer to the current head of the mailbox's internal circular buffer.
- void ∗ GetTailPointer (void)

  GetTailPointer Return a pointer to the current tail of the mailbox's internal circular buffer.
- void CopyData (const void ∗src_, void ∗dst_, uint16_t len_)

  CopyData Perform a direct byte-copy from a source to a destination object.
- void MoveTailForward (void)

  MoveTailForward Move the tail index forward one element.
- void MoveHeadForward (void)

  MoveHeadForward Move the head index forward one element.
- void MoveTailBackward (void)

  MoveTailBackward Move the tail index backward one element.
- void MoveHeadBackward (void)

  MoveHeadBackward Move the head index backward one element.
- bool Send_i (const void ∗pvData_, bool bTail_, uint32_t u32TimeoutMS_)

  Send_i Internal method which implements all Send() methods in the class.
- bool Receive_i (void ∗pvData_, bool bTail_, uint32_t u32WaitTimeMS_)

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

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.

19.18.1 Detailed Description

The Mailbox class. This class implements an IPC mechnism based on sending/receiving 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.

19.18.2 Constructor & Destructor Documentation

19.18.2.1 ∼Mailbox()

```
Mark3::Mailbox::∼Mailbox ( )
```

Definition at line 25 of file mailbox.cpp.

19.18.3 Member Function Documentation

19.18.3.1 CopyData()

```
void Mark3::Mailbox::CopyData (
            const void * src_,
            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 216 of file mailbox.h.

19.18.3.2 GetFreeSlots()

```
uint16_t Mark3::Mailbox::GetFreeSlots (
            void ) [inline]
```

Definition at line 170 of file mailbox.h.

19.18.3.3 GetHeadPointer()

```
void* Mark3::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 187 of file mailbox.h.

19.18.3.4 GetTailPointer()

```
void* Mark3::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 201 of file mailbox.h.

19.18.3.5 Init() [1/2]

```
void Mark3::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

Definition at line 34 of file mailbox.cpp.

19.18.3.6 Init() [2/2]

```
Mailbox * Mark3::Mailbox::Init (
            uint16_t u16BufferSize_,
            uint16_t u16ElementSize_ )  [static]
```

Init Create and initialize the mailbox object prior to its use. This must be called before any calls can be made to the object. This version of the API alloctes the buffer space from the kernel's Auto-Allocation heap, which cannot be returned back. As a result, this is only suitable for cases where the mailbox will be created once on startup, and persist for the duration of the system.

Parameters

*u16BufferSize_*	Size of the mailbox buffer, in bytes
*u16ElementSize_*	Size of each envelope, in bytes

Definition at line 59 of file mailbox.cpp.

19.18.3.7 IsEmpty()

```
bool Mark3::Mailbox::IsEmpty (
            void )  [inline]
```

Definition at line 177 of file mailbox.h.

19.18.3.8 IsFull()

```
bool Mark3::Mailbox::IsFull (
            void )  [inline]
```

Definition at line 176 of file mailbox.h.

19.18.3.9 MoveHeadBackward()

```
void Mark3::Mailbox::MoveHeadBackward (
            void )  [inline], [private]
```

MoveHeadBackward Move the head index backward one element.

Definition at line 263 of file mailbox.h.

19.18.3.10 MoveHeadForward()

```
void Mark3::Mailbox::MoveHeadForward (
            void ) [inline], [private]
```

MoveHeadForward Move the head index forward one element.

Definition at line 239 of file mailbox.h.

19.18.3.11 MoveTailBackward()

```
void Mark3::Mailbox::MoveTailBackward (
            void ) [inline], [private]
```

MoveTailBackward Move the tail index backward one element.

Definition at line 251 of file mailbox.h.

19.18.3.12 MoveTailForward()

```
void Mark3::Mailbox::MoveTailForward (
            void ) [inline], [private]
```

MoveTailForward Move the tail index forward one element.

Definition at line 227 of file mailbox.h.

19.18.3.13 operator new()

```
void* Mark3::Mailbox::operator new (
            size_t sz,
            void * pv ) [inline]
```

Definition at line 38 of file mailbox.h.

19.18.3.14 Receive() [1/2]

```
void Mark3::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

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

Examples:

lab11_mailboxes/main.cpp.

Definition at line 83 of file mailbox.cpp.

19.18.3.15 Receive() [2/2]

```
bool Mark3::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 90 of file mailbox.cpp.

19.18.3.16 Receive_i()

```
bool Mark3::Mailbox::Receive_i (
            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 197 of file mailbox.cpp.

19.18.3.17 ReceiveTail() [1/2]

```
void Mark3::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 97 of file mailbox.cpp.

19.18.3.18 ReceiveTail() [2/2]

```
bool Mark3::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.
*u32TimeoutMS_*	Maximum time to wait for delivery.

Returns

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

Definition at line 104 of file mailbox.cpp.

19.18.3.19 Send() [1/2]

```
bool Mark3::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

*pv↵ Data_*	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 111 of file mailbox.cpp.

19.18.3.20 Send() [2/2]

```
bool Mark3::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 125 of file mailbox.cpp.

19.18.3.21 Send_i()

```
bool Mark3::Mailbox::Send_i (
            const void * pvData_,
            bool bTail_,
            uint32_t u32TimeoutMS_ )  [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
*u32TimeoutM↩ S_*	Time to wait before timeout (in ms).

Returns

true - data successfully written, false - buffer full

Definition at line 139 of file mailbox.cpp.

19.18.3.22 SendTail() [1/2]

```
bool Mark3::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

*pv↩ Data_*	Pointer to the data object to send to the mailbox.

Returns

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

Definition at line 118 of file mailbox.cpp.

19.18.3.23 SendTail() [2/2]

```
bool Mark3::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 132 of file mailbox.cpp.

19.18.4 Member Data Documentation

19.18.4.1 m_clRecvSem

```
Semaphore Mark3::Mailbox::m_clRecvSem [private]
```

Counting semaphore used to synchronize threads on the object.

Definition at line 302 of file mailbox.h.

19.18.4.2 m_clSendSem

```
Semaphore Mark3::Mailbox::m_clSendSem [private]
```

Binary semaphore for send-blocked threads.

Definition at line 303 of file mailbox.h.

19.18.4.3 m_pvBuffer

```
const void* Mark3::Mailbox::m_pvBuffer [private]
```

Pointer to the data-buffer managed by this mailbox.

Definition at line 300 of file mailbox.h.

19.18.4.4 m_u16Count

```
uint16_t Mark3::Mailbox::m_u16Count [private]
```

Count of items in the mailbox.

Definition at line 296 of file mailbox.h.

19.18.4.5 m_u16ElementSize

```
uint16_t Mark3::Mailbox::m_u16ElementSize [private]
```

Size of the objects tracked in this mailbox.

Definition at line 299 of file mailbox.h.

19.18.4.6 m_u16Free

```
volatile uint16_t Mark3::Mailbox::m_u16Free [private]
```

Current number of free slots in the mailbox.

Definition at line 297 of file mailbox.h.

19.18.4.7 m_u16Head

```
uint16_t Mark3::Mailbox::m_u16Head [private]
```

Current head index.

Definition at line 293 of file mailbox.h.

19.18.4.8 m_u16Tail

```
uint16_t Mark3::Mailbox::m_u16Tail [private]
```

Current tail index.

Definition at line 294 of file mailbox.h.

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

- /home/moslevin/projects/m3-repo/kernel/src/public/mailbox.h
- /home/moslevin/projects/m3-repo/kernel/src/mailbox.cpp

19.19 Mark3::MemUtil Class Reference

String and Memory manipu32ation class.

```
#include <memutil.h>
```

Static Public Member Functions

- static void DecimalToHex (uint8_t u8Data_, char *szText_)

  DecimalToHex.
- static void DecimalToHex (uint16_t u16Data_, char *szText_)
- static void DecimalToHex (uint32_t u32Data_, char *szText_)
- static void DecimalToHex (uint64_t u64Data_, char *szText_)
- static void DecimalToString (uint8_t u8Data_, char *szText_)

  DecimalToString.
- static void DecimalToString (uint16_t u16Data_, char *szText_)
- static void DecimalToString (uint32_t u32Data_, char *szText_)
- static void DecimalToString (uint64_t u64Data_, char *szText_)
- static bool StringToDecimal8 (const char *szText_, uint8_t *pu8Out_)

  StringToDecimial8.
- static bool StringToDecimal16 (const char *szText_, uint16_t *pu16Out_)
- static bool StringToDecimal32 (const char *szText_, uint32_t *pu32Out_)
- static bool StringToDecimal64 (const char *szText_, uint64_t *pu64Out_)
- static uint8_t Checksum8 (const void *pvSrc_, uint16_t u16Len_)

  Checksum8.
- static uint16_t Checksum16 (const void *pvSrc_, uint16_t u16Len_)

  Checksum16.
- static uint16_t StringLength (const char *szStr_)

  StringLength.
- static bool CompareStrings (const char *szStr1_, const char *szStr2_)

  CompareStrings.
- static bool CompareStrings (const char *szStr1_, const char *szStr2_, uint16_t u16Length_)
- static void CopyMemory (void *pvDst_, const void *pvSrc_, uint16_t u16Len_)

  CopyMemory.
- static void CopyString (char *szDst_, const char *szSrc_)

  CopyString.
- static int16_t StringSearch (const char *szBuffer_, const char *szPattern_)

StringSearch.
- static bool CompareMemory (const void ∗pvMem1_, const void ∗pvMem2_, uint16_t u16Len_)

  CompareMemory.
- static void SetMemory (void ∗pvDst_, uint8_t u8Val_, uint16_t u16Len_)

  SetMemory.
- static uint8_t Tokenize (const char ∗szBuffer_, Token_t ∗pastTokens_, uint8_t u8MaxTokens_)

  Tokenize Function to tokenize a string based on a space delimeter. This is a non-destructive function, which popu32ates a Token_t descriptor array.

19.19.1 Detailed Description

String and Memory manipu32ation class.

Utility method class implementing common memory and string manipu32ation functions, without relying on an external standard library implementation which might not be available on some toolchains, may be closed source, or may not be thread-safe.

Definition at line 46 of file memutil.h.

19.19.2 Member Function Documentation

19.19.2.1 Checksum16()

```
uint16_t Mark3::MemUtil::Checksum16 (
            const void * pvSrc_,
            uint16_t u16Len_ )  [static]
```

Checksum16.

Compute the 16-bit addative checksum of a memory buffer.

Parameters

*pvSrc_*	Memory buffer to compute a 16-bit checksum of.
*u16Len_*	Length of the buffer in bytes.

Returns

16-bit checksum of the memory block.

Definition at line 346 of file memutil.cpp.

19.19.2.2 Checksum8()

```
uint8_t Mark3::MemUtil::Checksum8 (
             const void * pvSrc_,
             uint16_t u16Len_ )  [static]
```

Checksum8.

Compute the 8-bit addative checksum of a memory buffer.

Parameters

pvSrc←↩ _	Memory buffer to compute a 8-bit checksum of.
*u16←↩ Len_*	Length of the buffer in bytes.

Returns

8-bit checksum of the memory block.

Definition at line 333 of file memutil.cpp.

19.19.2.3 CompareMemory()

```
bool Mark3::MemUtil::CompareMemory (
             const void * pvMem1_,
             const void * pvMem2_,
             uint16_t u16Len_ )  [static]
```

CompareMemory.

Compare the contents of two memory buffers to eachother

Parameters

*pv←↩ Mem1_*	First buffer to compare
*pv←↩ Mem2_*	Second buffer to compare
u16Len←↩ _	Length of buffer (in bytes) to compare

Returns

true if the buffers match, false if they do not.

Definition at line 467 of file memutil.cpp.

19.19.2.4 CompareStrings() [1/2]

```
bool Mark3::MemUtil::CompareStrings (
            const char * szStr1_,
            const char * szStr2_ )  [static]
```

CompareStrings.

Compare the contents of two zero-terminated string buffers to eachother.

Parameters

*sz← Str1_*	First string to compare
*sz← Str2_*	Second string to compare

Returns

true if strings match, false otherwise.

Definition at line 372 of file memutil.cpp.

19.19.2.5 CompareStrings() [2/2]

```
bool Mark3::MemUtil::CompareStrings (
            const char * szStr1_,
            const char * szStr2_,
            uint16_t u16Length_ )  [static]
```

Definition at line 391 of file memutil.cpp.

19.19.2.6 CopyMemory()

```
void Mark3::MemUtil::CopyMemory (
            void * pvDst_,
            const void * pvSrc_,
            uint16_t u16Len_ )  [static]
```

CopyMemory.

Copy one buffer in memory into another.

Parameters

pvDst← _	Pointer to the destination buffer
pvSrc← _	Pointer to the source buffer
*u16← Len_*	Number of bytes to copy from source to destination

Definition at line 408 of file memutil.cpp.

19.19.2.7 CopyString()

```
void Mark3::MemUtil::CopyString (
            char * szDst_,
            const char * szSrc_ ) [static]
```

CopyString.

Copy a string from one buffer into another.

Parameters

*szDst_*	Pointer to the buffer to copy into
*szSrc_*	Pointer to the buffer to copy data from

Definition at line 422 of file memutil.cpp.

19.19.2.8 DecimalToHex() [1/4]

```
void Mark3::MemUtil::DecimalToHex (
            uint8_t u8Data_,
            char * szText_ ) [static]
```

DecimalToHex.

Convert an 8-bit unsigned binary value as a hexadecimal string.

Parameters

*u8Data_*	Value to convert into a string
*szText_*	Destination string buffer (3 bytes minimum)

Examples:

lab9_dynamic_threads/main.cpp.

Definition at line 29 of file memutil.cpp.

19.19.2.9 DecimalToHex() [2/4]

```
void Mark3::MemUtil::DecimalToHex (
            uint16_t u16Data_,
            char * szText_ ) [static]
```

Definition at line 55 of file memutil.cpp.

19.19.2.10 DecimalToHex() [3/4]

```
void Mark3::MemUtil::DecimalToHex (
            uint32_t u32Data_,
            char * szText_ ) [static]
```

Definition at line 81 of file memutil.cpp.

19.19.2.11 DecimalToHex() [4/4]

```
void Mark3::MemUtil::DecimalToHex (
            uint64_t u64Data_,
            char * szText_ ) [static]
```

Definition at line 107 of file memutil.cpp.

19.19.2.12 DecimalToString() [1/4]

```
void Mark3::MemUtil::DecimalToString (
            uint8_t u8Data_,
            char * szText_ ) [static]
```

DecimalToString.

Convert an 8-bit unsigned binary value as a decimal string.

Parameters

*u8Data_*	Value to convert into a string
*szText_*	Destination string buffer (4 bytes minimum)

Examples:

lab9_dynamic_threads/main.cpp.

Definition at line 133 of file memutil.cpp.

19.19.2.13 DecimalToString() [2/4]

```
void Mark3::MemUtil::DecimalToString (
            uint16_t u16Data_,
            char * szText_ ) [static]
```

Definition at line 157 of file memutil.cpp.

19.19.2.14 DecimalToString() [3/4]

```
void Mark3::MemUtil::DecimalToString (
            uint32_t u32Data_,
            char * szText_ ) [static]
```

Definition at line 178 of file memutil.cpp.

19.19.2.15 DecimalToString() [4/4]

```
void Mark3::MemUtil::DecimalToString (
            uint64_t u64Data_,
            char * szText_ ) [static]
```

Definition at line 199 of file memutil.cpp.

19.19.2.16 SetMemory()

```
void Mark3::MemUtil::SetMemory (
            void * pvDst_,
            uint8_t u8Val_,
            uint16_t u16Len_ ) [static]
```

SetMemory.

Initialize a buffer of memory to a specified 8-bit pattern

Parameters

*pvDst_*	Destination buffer to set
*u8Val_*	8-bit pattern to initialize each byte of destination with
*u16Len_*	Length of the buffer (in bytes) to initialize

Definition at line 486 of file memutil.cpp.

19.19.2.17 StringLength()

```
uint16_t Mark3::MemUtil::StringLength (
            const char * szStr_ ) [static]
```

StringLength.

Compute the length of a string in bytes.

Parameters

*sz←↩ Str_*	Pointer to the zero-terminated string to calculate the length of

Returns

length of the string (in bytes), not including the 0-terminator.

Definition at line 360 of file memutil.cpp.

19.19.2.18 StringSearch()

```
int16_t Mark3::MemUtil::StringSearch (
            const char * szBuffer_,
            const char * szPattern_ ) [static]
```

StringSearch.

Search for the presence of one string as a substring within another.

Parameters

szBuffer←↩ _	Buffer to search for pattern within
*sz←↩ Pattern_*	Pattern to search for in the buffer

Returns

Index of the first instance of the pattern in the buffer, or -1 on no match.

Definition at line 436 of file memutil.cpp.

19.19.2.19 StringToDecimal16()

```
bool Mark3::MemUtil::StringToDecimal16 (
             const char * szText_,
             uint16_t * pu16Out_ )  [static]
```

Definition at line 248 of file memutil.cpp.

19.19.2.20 StringToDecimal32()

```
bool Mark3::MemUtil::StringToDecimal32 (
             const char * szText_,
             uint32_t * pu32Out_ )  [static]
```

Definition at line 276 of file memutil.cpp.

19.19.2.21 StringToDecimal64()

```
bool Mark3::MemUtil::StringToDecimal64 (
             const char * szText_,
             uint64_t * pu64Out_ )  [static]
```

Definition at line 304 of file memutil.cpp.

19.19.2.22 StringToDecimal8()

```
bool Mark3::MemUtil::StringToDecimal8 (
             const char * szText_,
             uint8_t * pu8Out_ )  [static]
```

StringToDecimial8.

Convert a string to an unsigned integer value.

Parameters

*szText_*	String to convert
*pu8Out_*	Pointer to a uint8_t that will contain the result

Returns

true on success, false on invalid parameters or failure to convert the input string to an unsigned integer value

Definition at line 220 of file memutil.cpp.

19.19.2.23 Tokenize()

```
uint8_t Mark3::MemUtil::Tokenize (
            const char * szBuffer_,
            Token_t * pastTokens_,
            uint8_t u8MaxTokens_ ) [static]
```

Tokenize Function to tokenize a string based on a space delimeter. This is a non-destructive function, which popu32ates a Token_t descriptor array.

Parameters

*szBuffer_*	String to tokenize
*pastTokens_*	Pointer to the array of token descriptors
*u8Max←Tokens_*	Maximum number of tokens to parse (i.e. size of pastTokens_)

Returns

Count of tokens parsed

Definition at line 496 of file memutil.cpp.

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

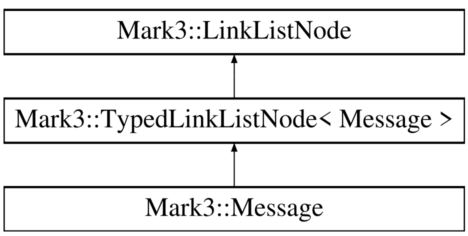
- /home/moslevin/projects/m3-repo/kernel/lib/memutil/public/memutil.h
- /home/moslevin/projects/m3-repo/kernel/lib/memutil/memutil.cpp

19.20 Mark3::Message Class Reference

the Message class. This object provides threadsafe message-based IPC services based on exchange of objects containing a data pointer and minimal application-defined metadata. Messages are to be allocated/produced by the sender, and deallocated/consumed by the receiver.

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

Inheritance diagram for Mark3::Message:



Public Member Functions

- void ∗ operator new (size_t sz, void ∗pv)
- void Init ()

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

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

  GetData Get the data pointer stored in the message upon receipt.
- void SetCode (uint16_t u16Code_)

  SetCode Set the code in the message before transmission.
- uint16_t GetCode ()

  GetCode Return the code set in the message upon receipt.

Private Attributes

- void ∗ m_pvData

  Pointer to the message data.
- uint16_t m_u16Code

  Message code, providing context for the message.

Additional Inherited Members

19.20.1 Detailed Description

the Message class. This object provides threadsafe message-based IPC services based on exchange of objects containing a data pointer and minimal application-defined metadata. Messages are to be allocated/produced by the sender, and deallocated/consumed by the receiver.

Examples:

lab8_messages/main.cpp.

Definition at line 97 of file message.h.

19.20.2 Member Function Documentation

19.20.2.1 GetCode()

```
uint16_t Mark3::Message::GetCode ( ) [inline]
```

GetCode Return the code set in the message upon receipt.

Returns

user code set in the object

Definition at line 139 of file message.h.

19.20.2.2 GetData()

```
void* Mark3::Message::GetData ( ) [inline]
```

GetData Get the data pointer stored in the message upon receipt.

Returns

Pointer to the data set in the message object

Definition at line 125 of file message.h.

19.20.2.3 Init()

```
void Mark3::Message::Init (
             void ) [inline]
```

Init Initialize the data and code in the message.

Definition at line 105 of file message.h.

19.20.2.4 operator new()

```
void* Mark3::Message::operator new (
             size_t sz,
             void * pv ) [inline]
```

Definition at line 100 of file message.h.

19.20.2.5 SetCode()

```
void Mark3::Message::SetCode (
             uint16_t u16Code_ ) [inline]
```

SetCode Set the code in the message before transmission.

Parameters

*u16←Code_*	Data code to set in the object

Examples:

lab8_messages/main.cpp.

Definition at line 132 of file message.h.

19.20.2.6 SetData()

```
void Mark3::Message::SetData (
            void * pvData_ )  [inline]
```

SetData Set the data pointer for the message before transmission.

Parameters

*pv←→ Data_*	Pointer to the data object to send in the message

Definition at line 118 of file message.h.

19.20.3 Member Data Documentation

19.20.3.1 m_pvData

```
void* Mark3::Message::m_pvData  [private]
```

Pointer to the message data.

Definition at line 143 of file message.h.

19.20.3.2 m_u16Code

```
uint16_t Mark3::Message::m_u16Code  [private]
```

Message code, providing context for the message.

Definition at line 146 of file message.h.

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

- /home/moslevin/projects/m3-repo/kernel/src/public/message.h

19.21 Mark3::MessagePool Class Reference

The MessagePool Class The MessagePool class implements a simple allocator for message objects exchanged between threads. The sender allocates (pop's) messages, then sends them to the receiver. Upon receipt, it is the receiver's responsibility to deallocate (push) the message back to the pool.

`#include <message.h>`

Public Member Functions

- void ∗ operator new (size_t sz, void ∗pv)
- ∼MessagePool ()
- void Init ()

  Init.
- void Push (Message ∗pclMessage_)

  Push.
- Message ∗ Pop ()

  Pop.
- Message ∗ GetHead ()

  GetHead.

Private Attributes

- TypedDoubleLinkList< Message > m_clList

  Linked list used to manage the Message objects.

19.21.1 Detailed Description

The MessagePool Class The MessagePool class implements a simple allocator for message objects exchanged between threads. The sender allocates (pop's) messages, then sends them to the receiver. Upon receipt, it is the receiver's responsibility to deallocate (push) the message back to the pool.

Examples:

lab8_messages/main.cpp.

Definition at line 157 of file message.h.

19.21.2 Constructor & Destructor Documentation

19.21.2.1 ∼MessagePool()

```
Mark3::MessagePool::~MessagePool ( ) [inline]
```

Definition at line 161 of file message.h.

19.21.3 Member Function Documentation

19.21.3.1 GetHead()

```
Message * Mark3::MessagePool::GetHead ( )
```

GetHead.

Return a pointer to the first element in the message list

Returns

Definition at line 52 of file message.cpp.

19.21.3.2 Init()

```
void Mark3::MessagePool::Init (
            void  )
```

Init.

Initialize the message queue prior to use

Definition at line 26 of file message.cpp.

19.21.3.3 operator new()

```
void* Mark3::MessagePool::operator new (
            size_t sz,
            void * pv )  [inline]
```

Definition at line 160 of file message.h.

19.21.3.4 Pop()

```
Message * Mark3::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

Examples:

lab8_messages/main.cpp.

Definition at line 41 of file message.cpp.

19.21.3.5 Push()

```
void Mark3::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

*pcl↵ Message_*	Pointer to the Message object to return back to the queue

Examples:

lab8_messages/main.cpp.

Definition at line 32 of file message.cpp.

19.21.4 Member Data Documentation

19.21.4.1 m_clList

`TypedDoubleLinkList<Message> Mark3::MessagePool::m_clList [private]`

Linked list used to manage the Message objects.

Definition at line 201 of file message.h.

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

- /home/moslevin/projects/m3-repo/kernel/src/public/message.h
- /home/moslevin/projects/m3-repo/kernel/src/message.cpp

19.22 Mark3::MessageQueue Class Reference

The MessageQueue class. Implements a mechanism used to send/receive data between threads. Allows threads to block, waiting for messages to be sent from other contexts.

`#include <message.h>`

Public Member Functions

- void ∗ operator new (size_t sz, void ∗pv)
- ∼MessageQueue ()
- 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.
- TypedDoubleLinkList< Message > m_clLinkList

  List object used to store messages.

19.22.1 Detailed Description

The MessageQueue class. Implements a mechanism used to send/receive data between threads. Allows threads to block, waiting for messages to be sent from other contexts.

Examples:

lab8_messages/main.cpp.

Definition at line 210 of file message.h.

19.22.2 Constructor & Destructor Documentation

19.22.2.1 ∼MessageQueue()

```
Mark3::MessageQueue::∼MessageQueue ( )  [inline]
```

Definition at line 214 of file message.h.

19.22.3 Member Function Documentation

19.22.3.1 GetCount()

```
uint16_t Mark3::MessageQueue::GetCount ( )
```

GetCount.

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

Returns

Count of pending messages in the queue.

Definition at line 108 of file message.cpp.

19.22.3.2 Init()

```
void Mark3::MessageQueue::Init (
             void  )
```

Init.

Initialize the message queue prior to use.

Definition at line 58 of file message.cpp.

19.22.3.3 operator new()

```
void* Mark3::MessageQueue::operator new (
            size_t sz,
            void * pv )  [inline]
```

Definition at line 213 of file message.h.

19.22.3.4 Receive() [1/2]

```
Message * Mark3::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 64 of file message.cpp.

19.22.3.5 Receive() [2/2]

```
Message * Mark3::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 nullptr.

Parameters

*u32TimeWaitMS_*	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 nullptr on timeout.

Definition at line 70 of file message.cpp.

19.22.3.6 Receive_i()

```
Message * Mark3::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 76 of file message.cpp.

19.22.3.7 Send()

```
void Mark3::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

*pcl↩ Src_*	Pointer to the message object to add to the queue

Examples:

lab8_messages/main.cpp.

Definition at line 92 of file message.cpp.

19.22.4 Member Data Documentation

19.22.4.1 m_clLinkList

`TypedDoubleLinkList<Message> Mark3::MessageQueue::m_clLinkList [private]`

List object used to store messages.

Definition at line 284 of file message.h.

19.22.4.2 m_clSemaphore

`Semaphore Mark3::MessageQueue::m_clSemaphore [private]`

Counting semaphore used to manage thread blocking.

Definition at line 281 of file message.h.

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

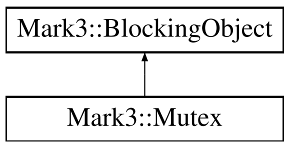
- /home/moslevin/projects/m3-repo/kernel/src/public/message.h
- /home/moslevin/projects/m3-repo/kernel/src/message.cpp

19.23 Mark3::Mutex Class Reference

The Mutex Class. Class providing Mutual-exclusion locks, based on BlockingObject.

`#include <mutex.h>`

Inheritance diagram for Mark3::Mutex:



Public Member Functions

- void ∗ operator new (size_t sz, void ∗pv)
- ∼Mutex ()
- void Init (bool bRecursive_=true)

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

  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.
- bool Claim (uint32_t u32WaitTimeMS_)

  Claim Claim a mutex, with timeout.
- void 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.
- void Release ()

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

Private Member Functions

- uint8_t WakeNext ()
  WakeNext.
- bool Claim_i (uint32_t u32WaitTimeMS_)
  Claim_i Abstracts out timed/non-timed mutex claim operations.

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.
- bool m_bRecursive
  Whether or not the lock is recursive.
- PORT_PRIO_TYPE m_uMaxPri
  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

19.23.1 Detailed Description

The Mutex Class. Class providing Mutual-exclusion locks, based on BlockingObject.

Examples:

lab5_mutexes/main.cpp.

Definition at line 63 of file mutex.h.

19.23.2 Constructor & Destructor Documentation

19.23.2.1 ∼Mutex()

```
Mark3::Mutex::∼Mutex ( )
```

Definition at line 55 of file mutex.cpp.

19.23.3 Member Function Documentation

19.23.3.1 Claim() [1/2]

```
void Mark3::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 185 of file mutex.cpp.

19.23.3.2 Claim() [2/2]

```
bool Mark3::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 191 of file mutex.cpp.

19.23.3.3 Claim_i()

```
bool Mark3::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 108 of file mutex.cpp.

19.23.3.4 Init()

```
void Mark3::Mutex::Init (
            bool bRecursive_ = true )
```

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

Parameters

b↩ Recursive↩ _	Whether or not the mutex can be recursively locked.

Definition at line 93 of file mutex.cpp.

19.23.3.5 operator new()

```
void* Mark3::Mutex::operator new (
            size_t sz,
            void * pv ) [inline]
```

Definition at line 66 of file mutex.h.

19.23.3.6 Release()

```
void Mark3::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 197 of file mutex.cpp.

19.23.3.7 WakeMe()

```
void Mark3::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 65 of file mutex.cpp.

19.23.3.8 WakeNext()

```
uint8_t Mark3::Mutex::WakeNext ( ) [private]
```

WakeNext.

Wake the next thread waiting on the Mutex.

Definition at line 73 of file mutex.cpp.

19.23.4 Member Data Documentation

19.23.4.1 m_bReady

```
bool Mark3::Mutex::m_bReady [private]
```

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

Definition at line 159 of file mutex.h.

19.23.4.2 m_bRecursive

```
bool Mark3::Mutex::m_bRecursive [private]
```

Whether or not the lock is recursive.

Definition at line 160 of file mutex.h.

19.23.4.3 m_pclOwner

```
Thread* Mark3::Mutex::m_pclOwner [private]
```

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

Definition at line 162 of file mutex.h.

19.23.4.4 m_u8Recurse

```
uint8_t Mark3::Mutex::m_u8Recurse [private]
```

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

Definition at line 158 of file mutex.h.

19.23.4.5 m_uMaxPri

```
PORT_PRIO_TYPE Mark3::Mutex::m_uMaxPri [private]
```

Maximum priority of thread in queue, used for priority inheritence.

Definition at line 161 of file mutex.h.

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

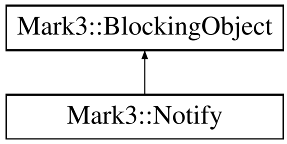
- /home/moslevin/projects/m3-repo/kernel/src/public/mutex.h
- /home/moslevin/projects/m3-repo/kernel/src/mutex.cpp

19.24 Mark3::Notify Class Reference

The Notify class. This class provides 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 Mark3::Notify:



Public Member Functions

- void ∗ operator new (size_t sz, void ∗pv)
- ∼Notify ()
- void Init (void)

    Init Initialze the Notification object prior to use.
- void 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.
- void Wait (bool ∗pbFlag_)

    Wait Block the current thread, waiting for a signal on the object.
- bool Wait (uint32_t u32WaitTimeMS_, bool ∗pbFlag_)

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

Private Attributes

- bool m_bPending

Additional Inherited Members

19.24.1 Detailed Description

The Notify class. This class provides 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.

19.24.2 Constructor & Destructor Documentation

19.24.2.1 ∼Notify()

```
Mark3::Notify::∼Notify ( )
```

Definition at line 47 of file notify.cpp.

19.24.3 Member Function Documentation

19.24.3.1 Init()

```
void Mark3::Notify::Init (
            void )
```

Init Initialze the Notification object prior to use.

Definition at line 57 of file notify.cpp.

19.24.3.2 operator new()

```
void* Mark3::Notify::operator new (
            size_t sz,
            void * pv ) [inline]
```

Definition at line 36 of file notify.h.

19.24.3.3 Signal()

```
void Mark3::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 66 of file notify.cpp.

19.24.3.4 Wait() [1/2]

```
void Mark3::Notify::Wait (
            bool * pbFlag_ )
```

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

Parameters

*pb←↩ Flag_*	Flag set to false on block, and true upon wakeup.

Examples:

lab10_notifications/main.cpp.

Definition at line 95 of file notify.cpp.

19.24.3.5 Wait() [2/2]

```
bool Mark3::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 125 of file notify.cpp.

19.24.3.6 WakeMe()

```
void Mark3::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

*pclChosen←↩ One_*	Thread to wake up

Definition at line 175 of file notify.cpp.

19.24.4 Member Data Documentation

19.24.4.1 m_bPending

```
bool Mark3::Notify::m_bPending [private]
```

Definition at line 89 of file notify.h.

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

- /home/moslevin/projects/m3-repo/kernel/src/public/notify.h
- /home/moslevin/projects/m3-repo/kernel/src/notify.cpp

19.25 Mark3::PriorityMapL1< T, C > Class Template Reference

The PriorityMapL1 class This class implements a priority bitmap data structure. Each bit in the objects internal storage represents a priority. When a bit is set, it indicates that something is scheduled at the bit's corresponding priority, when a bit is clear it indicates that no entities are scheduled at that priority. This object provides the fundamental logic required to implement efficient priority-based scheduling for the thread + coroutine schedulers in the kernel.

```
#include <priomapl1.h>
```

Public Member Functions

- PriorityMapL1 ()

  PriorityMap Initialize the priority map object, clearing the bitamp data to all 0's.
- void Set (T uXPrio_)

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

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

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

Static Private Member Functions

- static T PrioBit (T prio)
- static T PrioMapWordIndex (T prio)
- static T PriorityFromBitmap (T uXPrio_)

Private Attributes

- T m_uXPriorityMap

Static Private Attributes

- static constexpr size_t m_uXPrioMapShiftLUT [9] = {0, 3, 4, 0, 5, 0, 0, 0, 6}
- static constexpr auto m_uXPrioMapWordShift = T { m_uXPrioMapShiftLUT[sizeof(T)] }
- static constexpr auto m_uXPrioMapBits = T { 8 ∗ sizeof(T) }
- static constexpr auto m_uXPrioMapBitMask = T { (1 << m_uXPrioMapWordShift) - 1 }

19.25.1 Detailed Description

template<typename T, size_t C>
class Mark3::PriorityMapL1< T, C >

The PriorityMapL1 class This class implements a priority bitmap data structure. Each bit in the objects internal storage represents a priority. When a bit is set, it indicates that something is scheduled at the bit's corresponding priority, when a bit is clear it indicates that no entities are scheduled at that priority. This object provides the fundamental logic required to implement efficient priority-based scheduling for the thread + coroutine schedulers in the kernel.

The L1 version of the datastructure uses a single unsigned integer (specified by the templated typname "T"), to support "C" priorities. Thus, the maximum number of priorities supported in the map is the number of bits in the "T" parameter.

Definition at line 45 of file priomapl1.h.

19.25.2 Constructor & Destructor Documentation

19.25.2.1 PriorityMapL1()

```
template<typename T , size_t C>
Mark3::PriorityMapL1< T, C >::PriorityMapL1 ( )  [inline]
```

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

Definition at line 52 of file priomapl1.h.

19.25.3 Member Function Documentation

19.25.3.1 Clear()

```
template<typename T , size_t C>
void Mark3::PriorityMapL1< T, C >::Clear (
             T uXPrio_ )  [inline]
```

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

Parameters

*uX↩ Prio_*	Priority level to clear the bitmap data for.

Definition at line 73 of file priomapl1.h.

19.25.3.2 HighestPriority()

```
template<typename T , size_t C>
T Mark3::PriorityMapL1< T, C >::HighestPriority (
            void ) [inline]
```

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 86 of file priomapl1.h.

19.25.3.3 PrioBit()

```
template<typename T , size_t C>
static T Mark3::PriorityMapL1< T, C >::PrioBit (
            T prio ) [inline], [static], [private]
```

Definition at line 93 of file priomapl1.h.

19.25.3.4 PrioMapWordIndex()

```
template<typename T , size_t C>
static T Mark3::PriorityMapL1< T, C >::PrioMapWordIndex (
            T prio ) [inline], [static], [private]
```

Definition at line 95 of file priomapl1.h.

19.25.3.5 PriorityFromBitmap()

```
template<typename T , size_t C>
static T Mark3::PriorityMapL1< T, C >::PriorityFromBitmap (
            T uXPrio_ ) [inline], [static], [private]
```

Definition at line 97 of file priomapl1.h.

19.25.3.6 Set()

```
template<typename T , size_t C>
void Mark3::PriorityMapL1< T, C >::Set (
            T uXPrio_ ) [inline]
```

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

Parameters

*uX↩ Prio_*	Priority level to set the bitmap data for.

Definition at line 62 of file priomapl1.h.

19.25.4 Member Data Documentation

19.25.4.1 m_uXPrioMapBitMask

```
template<typename T , size_t C>
constexpr auto Mark3::PriorityMapL1< T, C >::m_uXPrioMapBitMask = T { (1 << m_uXPrioMapWord↩
Shift) - 1 }  [static], [private]
```

Definition at line 122 of file priomapl1.h.

19.25.4.2 m_uXPrioMapBits

```
template<typename T , size_t C>
constexpr auto Mark3::PriorityMapL1< T, C >::m_uXPrioMapBits = T { 8 * sizeof(T) }  [static],
[private]
```

Definition at line 121 of file priomapl1.h.

19.25.4.3 m_uXPrioMapShiftLUT

```
template<typename T , size_t C>
constexpr size_t Mark3::PriorityMapL1< T, C >::m_uXPrioMapShiftLUT[9] = {0, 3, 4, 0, 5, 0, 0,
0, 6}  [static], [private]
```

Definition at line 119 of file priomapl1.h.

19.25.4.4 m_uXPrioMapWordShift

```
template<typename T , size_t C>
constexpr auto Mark3::PriorityMapL1< T, C >::m_uXPrioMapWordShift = T { m_uXPrioMapShiftL↩
UT[sizeof(T)] }  [static], [private]
```

Definition at line 120 of file priomapl1.h.

19.25.4.5 m_uXPriorityMap

```
template<typename T , size_t C>
T Mark3::PriorityMapL1< T, C >::m_uXPriorityMap [private]
```

Definition at line 124 of file priomapl1.h.

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

- /home/moslevin/projects/m3-repo/kernel/src/public/priomapl1.h

19.26 Mark3::PriorityMapL2< T, C > Class Template Reference

The PriorityMapL2 class This class implements a priority bitmap data structure. Each bit in the objects internal storage represents a priority. When a bit is set, it indicates that something is scheduled at the bit's corresponding priority, when a bit is clear it indicates that no entities are scheduled at that priority. This object provides the fundamental logic required to implement efficient priority-based scheduling for the thread + coroutine schedulers in the kernel.

```
#include <priomapl2.h>
```

Public Member Functions

- PriorityMapL2 ()

  PriorityMap Initialize the priority map object, clearing the bitamp data to all 0's.
- void Set (T uXPrio_)

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

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

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

Static Private Member Functions

- static T PrioBit (T prio)
- static T PrioMapWordIndex (T prio)
- static T PriorityFromBitmap (T uXPrio_)

Private Attributes

- T m_auXPriorityMap [m_uXPrioMapNumWords]
- T m_uXPriorityMapL2

Static Private Attributes

- static constexpr size_t m_uXPrioMapShiftLUT [9] = {0, 3, 4, 0, 5, 0, 0, 0, 6}
- static constexpr auto m_uXPrioMapWordShift = T { m_uXPrioMapShiftLUT[sizeof(T)] }
- static constexpr auto m_uXPrioMapBits = T { 8 * sizeof(T) }
- static constexpr auto m_uXPrioMapBitMask = T { (1 << m_uXPrioMapWordShift) - 1 }
- static constexpr auto m_uXPrioMapNumWords = T { (C + (m_uXPrioMapBits - 1)) / m_uXPrioMapBits }

19.26.1 Detailed Description

template<typename T, size_t C>
class Mark3::PriorityMapL2< T, C >

The PriorityMapL2 class This class implements a priority bitmap data structure. Each bit in the objects internal storage represents a priority. When a bit is set, it indicates that something is scheduled at the bit's corresponding priority, when a bit is clear it indicates that no entities are scheduled at that priority. This object provides the fundamental logic required to implement efficient priority-based scheduling for the thread + coroutine schedulers in the kernel.

The L2 version of the datastructure uses a two-layer nested map structure, where a first layer bitmap contains a single unsigned integer of type "T", where each bit corresponds to an array entry in the second layer table. The second layer consists of an array of unsigned integers of type "T", where each bit in each element corresponds to a level of priority supported by the map structure. As a result, the maximum number of priorities ("C") supported by the object is n∗n, where n is the number of bits in the "T" parameter. i.e., if an 8-bit unsigned int is used, this object supports up to 64 priorities; and if a 32-bit unsigned int is used, the object supports up to 1024 priorities.

Definition at line 50 of file priomapl2.h.

19.26.2 Constructor & Destructor Documentation

19.26.2.1 PriorityMapL2()

```
template<typename T , size_t C>
Mark3::PriorityMapL2< T, C >::PriorityMapL2 ( )  [inline]
```

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

Definition at line 57 of file priomapl2.h.

19.26.3 Member Function Documentation

19.26.3.1 Clear()

```
template<typename T , size_t C>
void Mark3::PriorityMapL2< T, C >::Clear (
            T uXPrio_ )  [inline]
```

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

Parameters

*uX←Prio_*	Priority level to clear the bitmap data for.

Definition at line 82 of file priomapl2.h.

19.26.3.2 HighestPriority()

```
template<typename T , size_t C>
T Mark3::PriorityMapL2< T, C >::HighestPriority (
            void ) [inline]
```

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 100 of file priomapl2.h.

19.26.3.3 PrioBit()

```
template<typename T , size_t C>
static T Mark3::PriorityMapL2< T, C >::PrioBit (
            T prio ) [inline], [static], [private]
```

Definition at line 113 of file priomapl2.h.

19.26.3.4 PrioMapWordIndex()

```
template<typename T , size_t C>
static T Mark3::PriorityMapL2< T, C >::PrioMapWordIndex (
            T prio ) [inline], [static], [private]
```

Definition at line 115 of file priomapl2.h.

19.26.3.5 PriorityFromBitmap()

```
template<typename T , size_t C>
static T Mark3::PriorityMapL2< T, C >::PriorityFromBitmap (
            T uXPrio_ ) [inline], [static], [private]
```

Definition at line 117 of file priomapl2.h.

19.26.3.6 Set()

```
template<typename T , size_t C>
void Mark3::PriorityMapL2< T, C >::Set (
            T uXPrio_ ) [inline]
```

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

Parameters

*uX←↩ Prio_*	Priority level to set the bitmap data for.

Definition at line 68 of file priomapl2.h.

19.26.4 Member Data Documentation

19.26.4.1 m_auXPriorityMap

```
template<typename T , size_t C>
T Mark3::PriorityMapL2< T, C >::m_auXPriorityMap[m_uXPrioMapNumWords]  [private]
```

Definition at line 148 of file priomapl2.h.

19.26.4.2 m_uXPrioMapBitMask

```
template<typename T , size_t C>
constexpr auto Mark3::PriorityMapL2< T, C >::m_uXPrioMapBitMask = T { (1 << m_uXPrioMapWord←↩
Shift) - 1 }  [static], [private]
```

Definition at line 142 of file priomapl2.h.

19.26.4.3 m_uXPrioMapBits

```
template<typename T , size_t C>
constexpr auto Mark3::PriorityMapL2< T, C >::m_uXPrioMapBits = T { 8 * sizeof(T) }  [static],
[private]
```

Definition at line 141 of file priomapl2.h.

19.26.4.4 m_uXPrioMapNumWords

```
template<typename T , size_t C>
constexpr auto Mark3::PriorityMapL2< T, C >::m_uXPrioMapNumWords = T { (C + (m_uXPrioMapBits -
1)) / m_uXPrioMapBits }  [static], [private]
```

Definition at line 146 of file priomapl2.h.

19.26.4.5 m_uXPrioMapShiftLUT

```
template<typename T , size_t C>
constexpr size_t Mark3::PriorityMapL2< T, C >::m_uXPrioMapShiftLUT[9] = {0, 3, 4, 0, 5, 0, 0,
0, 6} [static], [private]
```

Definition at line 139 of file priomapl2.h.

19.26.4.6 m_uXPrioMapWordShift

```
template<typename T , size_t C>
constexpr auto Mark3::PriorityMapL2< T, C >::m_uXPrioMapWordShift = T { m_uXPrioMapShiftL←
UT[sizeof(T)] } [static], [private]
```

Definition at line 140 of file priomapl2.h.

19.26.4.7 m_uXPriorityMapL2

```
template<typename T , size_t C>
T Mark3::PriorityMapL2< T, C >::m_uXPriorityMapL2 [private]
```

Definition at line 149 of file priomapl2.h.

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

- /home/moslevin/projects/m3-repo/kernel/src/public/priomapl2.h

19.27 Mark3::ProfileTimer Class Reference

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.

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

Public Member Functions

- void Init ()
  Init Initialize the profiling timer prior to use. Can also be used to reset a timer that's been used previously.
- void Start ()
  Start Start a profiling session, if the timer is not already active. Has no effect if the timer is already active.
- void Stop ()
  Stop Stop the current profiling session, adding to the cumulative time for this timer, and the total iteration count.
- uint32_t GetAverage ()
  GetAverage Get the average time associated with this operation.
- uint32_t GetCurrent ()
  GetCurrent Return the current tick count held by the profiler. Valid for both active and stopped timers.

Private Attributes

- uint32_t m_u32StartTicks

    Cumulative tick-count for this timer.
- uint32_t m_u32CurrentIteration

    Tick count for current iteration.
- uint32_t m_u32Cumulative

    Cumulative ticks tracked.
- uint16_t m_u16Iterations

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

    Wheter or not the timer is active or stopped.

19.27.1 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 67 of file profile.h.

19.27.2 Member Function Documentation

19.27.2.1 GetAverage()

`uint32_t Mark3::ProfileTimer::GetAverage ( )`

GetAverage Get the average time associated with this operation.

Returns

Average tick count normalized over all iterations

Definition at line 63 of file profile.cpp.

19.27.2.2 GetCurrent()

`uint32_t Mark3::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 72 of file profile.cpp.

19.27.2.3 Init()

```
void Mark3::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 25 of file profile.cpp.

19.27.2.4 Start()

```
void Mark3::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 34 of file profile.cpp.

19.27.2.5 Stop()

```
void Mark3::ProfileTimer::Stop (
            void )
```

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

Definition at line 46 of file profile.cpp.

19.27.3 Member Data Documentation

19.27.3.1 m_bActive

```
bool Mark3::ProfileTimer::m_bActive [private]
```

Wheter or not the timer is active or stopped.

Definition at line 113 of file profile.h.

19.27.3.2 m_u16Iterations

`uint16_t Mark3::ProfileTimer::m_u16Iterations [private]`

Number of iterations executed for this profiling timer.

Definition at line 112 of file profile.h.

19.27.3.3 m_u32Cumulative

`uint32_t Mark3::ProfileTimer::m_u32Cumulative [private]`

Cumulative ticks tracked.

Definition at line 111 of file profile.h.

19.27.3.4 m_u32CurrentIteration

`uint32_t Mark3::ProfileTimer::m_u32CurrentIteration [private]`

Tick count for current iteration.

Definition at line 110 of file profile.h.

19.27.3.5 m_u32StartTicks

`uint32_t Mark3::ProfileTimer::m_u32StartTicks [private]`

Cumulative tick-count for this timer.

Definition at line 109 of file profile.h.

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

- /home/moslevin/projects/m3-repo/kernel/src/public/profile.h
- /home/moslevin/projects/m3-repo/kernel/src/profile.cpp

19.28 Mark3::Quantum Class Reference

The Quantum Class. Static-class used to implement Thread quantum functionality, which is fundamental to round-robin thread scheduling.

`#include <quantum.h>`

Static Public Member Functions

- static void Init ()
- static void SetInTimer ()

  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.
- static void ClearInTimer ()

  ClearInTimer Clear the flag once the timer callback function has been completed.
- static void Update (Thread *pclTargetThread_)

  Update Update the current thread being tracked for round-robing scheduling. Note - this has no effect if called from the Timer thread, or if the Timer thread is active.
- static void SetTimerThread (Thread *pclTimerThread_)

  SetTimerThread Pass the timer thread's Thread pointer to the Quantum module to track against requests to update the round-robin timer.
- static void Cancel ()

Static Private Attributes

- static Thread * m_pclActiveThread
- static Thread * m_pclTimerThread
- static uint16_t m_u16TicksRemain
- static bool m_bInTimer

19.28.1 Detailed Description

The Quantum Class. Static-class used to implement Thread quantum functionality, which is fundamental to round-robin thread scheduling.

Definition at line 42 of file quantum.h.

19.28.2 Member Function Documentation

19.28.2.1 Cancel()

```
static void Mark3::Quantum::Cancel ( )  [static]
```

Cancel the round-robin timer.

19.28.2.2 ClearInTimer()

```
static void Mark3::Quantum::ClearInTimer ( )  [static]
```

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

19.28.2.3 Init()

```
static void Mark3::Quantum::Init ( ) [static]
```

19.28.2.4 SetInTimer()

```
static void Mark3::Quantum::SetInTimer ( ) [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.

19.28.2.5 SetTimerThread()

```
static void Mark3::Quantum::SetTimerThread (
            Thread * pclTimerThread_ ) [inline], [static]
```

SetTimerThread Pass the timer thread's Thread pointer to the Quantum module to track against requests to update the round-robin timer.

Parameters

*pclTimerThread_*	Pointer to the Timer thread's Thread object.

Definition at line 79 of file quantum.h.

19.28.2.6 Update()

```
static void Mark3::Quantum::Update (
            Thread * pclTargetThread_ ) [static]
```

Update Update the current thread being tracked for round-robing scheduling. Note - this has no effect if called from the Timer thread, or if the Timer thread is active.

Parameters

*pclTargetThread_*	New thread to track.

19.28.3 Member Data Documentation

19.28.3.1 m_bInTimer

`bool Mark3::Quantum::m_bInTimer [static], [private]`

Definition at line 90 of file quantum.h.

19.28.3.2 m_pclActiveThread

`Thread* Mark3::Quantum::m_pclActiveThread [static], [private]`

Definition at line 87 of file quantum.h.

19.28.3.3 m_pclTimerThread

`Thread* Mark3::Quantum::m_pclTimerThread [static], [private]`

Definition at line 88 of file quantum.h.

19.28.3.4 m_u16TicksRemain

`uint16_t Mark3::Quantum::m_u16TicksRemain [static], [private]`

Definition at line 89 of file quantum.h.

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

- /home/moslevin/projects/m3-repo/kernel/src/public/quantum.h

19.29 Mark3::ReaderWriterLock Class Reference

The ReaderWriterLock class. This class implements an object that marshalls access to a resource based on the intended usage of the resource. A reader-writer lock permits multiple concurrent read access, or single-writer access to a resource. If the object holds a write lock, other writers, and all readers will block until the writer is finished. If the object holds reader locks, all writers will block until all readers are finished before the first writer can take ownership of the resource. This is based upon lower-level synchronization primatives, and is somewhat more heavyweight than primative synchronization types.

`#include <readerwriter.h>`

Public Member Functions

- void ∗ operator new (size_t sz, void ∗pv)
- void Init ()

  Init Initialize the reader-writer lock before use. Must be called before attempting any other operations on the object.
- void AcquireReader ()

  AcquireReader Acquire the object's reader lock. Multiple concurrent readers are allowed. If the writer lock is currently held, the calling thread will wait until the writer lock is relinquished.
- bool AcquireReader (uint32_t u32TimeoutMs_)

  AcquireReader Acquire the object's reader lock. Multiple concurrent readers are allowed. If the writer lock is currently held, the calling thread will wait until the writer lock is relinquished.
- void ReleaseReader ()

  ReleaseReader Release a previously-held reader lock.
- void AcquireWriter ()

  AcquireWriter Acquire the writer lock. Only a single writer is allowed to access the object at a time. This will block the currently-runnign thread until all other readers/writers have released their locks.
- bool AcquireWriter (uint32_t u32TimeoutMs_)

  AcquireWriter Acquire the writer lock. Only a single writer is allowed to access the object at a time. This will block the currently-runnign thread until all other readers/writers have released their locks.
- void ReleaseWriter ()

  ReleaseWriter Release the currently held writer, allowing other readers/writers to access the object.

Private Member Functions

- bool AcquireReader_i (uint32_t u32TimeoutMs_)

  AcquireReader_i Internal helper function for AcquireReaer.
- bool AcquireWriter_i (uint32_t u32TimeoutMs_)

  AcquireWriter_i Internal helper function for AcquireWriter.

Private Attributes

- Mutex m_clGlobalMutex

  Mutex used to lock the object against concurrent read + write.
- Mutex m_clReaderMutex

  Mutex used to lock object for readers.
- uint8_t m_u8ReadCount

  Number of concurrent readers.

19.29.1 Detailed Description

The ReaderWriterLock class. This class implements an object that marshalls access to a resource based on the intended usage of the resource. A reader-writer lock permits multiple concurrent read access, or single-writer access to a resource. If the object holds a write lock, other writers, and all readers will block until the writer is finished. If the object holds reader locks, all writers will block until all readers are finished before the first writer can take ownership of the resource. This is based upon lower-level synchronization primatives, and is somewhat more heavyweight than primative synchronization types.

Definition at line 40 of file readerwriter.h.

19.29.2 Member Function Documentation

19.29.2.1 AcquireReader() [1/2]

```
void Mark3::ReaderWriterLock::AcquireReader ( )
```

AcquireReader Acquire the object's reader lock. Multiple concurrent readers are allowed. If the writer lock is currently held, the calling thread will wait until the writer lock is relinquished.

Definition at line 32 of file readerwriter.cpp.

19.29.2.2 AcquireReader() [2/2]

```
bool Mark3::ReaderWriterLock::AcquireReader (
            uint32_t u32TimeoutMs_ )
```

AcquireReader Acquire the object's reader lock. Multiple concurrent readers are allowed. If the writer lock is currently held, the calling thread will wait until the writer lock is relinquished.

Parameters

*u32TimeoutMs_*	Maximum time to wait (in ms) before the operation is aborted

Returns

true on success, false on timeout

Definition at line 38 of file readerwriter.cpp.

19.29.2.3 AcquireReader_i()

```
bool Mark3::ReaderWriterLock::AcquireReader_i (
            uint32_t u32TimeoutMs_ ) [private]
```

AcquireReader_i Internal helper function for AcquireReaer.

Parameters

*u32TimeoutMs_*	Maximum time to wait (in ms) before the operation is aborted

Returns

true on success, false on timeout

Definition at line 73 of file readerwriter.cpp.

19.29.2.4 AcquireWriter() [1/2]

```
void Mark3::ReaderWriterLock::AcquireWriter ( )
```

AcquireWriter Acquire the writer lock. Only a single writer is allowed to access the object at a time. This will block the currently-runnign thread until all other readers/writers have released their locks.

Definition at line 55 of file readerwriter.cpp.

19.29.2.5 AcquireWriter() [2/2]

```
bool Mark3::ReaderWriterLock::AcquireWriter (
            uint32_t u32TimeoutMs_ )
```

AcquireWriter Acquire the writer lock. Only a single writer is allowed to access the object at a time. This will block the currently-runnign thread until all other readers/writers have released their locks.

Parameters

*u32TimeoutMs_*	Maximum time to wait (in ms) before the operation is aborted

Returns

true on success, false on timeout

Definition at line 61 of file readerwriter.cpp.

19.29.2.6 AcquireWriter_i()

```
bool Mark3::ReaderWriterLock::AcquireWriter_i (
            uint32_t u32TimeoutMs_ )  [private]
```

AcquireWriter_i Internal helper function for AcquireWriter.

Parameters

*u32Timeout↩ Ms_*	Maximum time to wait (in ms) before the operation is aborted

Returns

true on success, false on timeout

Definition at line 90 of file readerwriter.cpp.

19.29.2.7 Init()

```
void Mark3::ReaderWriterLock::Init (
            void )
```

Init Initialize the reader-writer lock before use. Must be called before attempting any other operations on the object.

Definition at line 24 of file readerwriter.cpp.

19.29.2.8 operator new()

```
void* Mark3::ReaderWriterLock::operator new (
            size_t sz,
            void * pv ) [inline]
```

Definition at line 43 of file readerwriter.h.

19.29.2.9 ReleaseReader()

```
void Mark3::ReaderWriterLock::ReleaseReader ( )
```

ReleaseReader Release a previously-held reader lock.

Definition at line 44 of file readerwriter.cpp.

19.29.2.10 ReleaseWriter()

```
void Mark3::ReaderWriterLock::ReleaseWriter ( )
```

ReleaseWriter Release the currently held writer, allowing other readers/writers to access the object.

Definition at line 67 of file readerwriter.cpp.

19.29.3 Member Data Documentation

19.29.3.1 m_clGlobalMutex

`Mutex Mark3::ReaderWriterLock::m_clGlobalMutex [private]`

Mutex used to lock the object against concurrent read + write.

Definition at line 116 of file readerwriter.h.

19.29.3.2 m_clReaderMutex

`Mutex Mark3::ReaderWriterLock::m_clReaderMutex [private]`

Mutex used to lock object for readers.

Definition at line 117 of file readerwriter.h.

19.29.3.3 m_u8ReadCount

`uint8_t Mark3::ReaderWriterLock::m_u8ReadCount [private]`

Number of concurrent readers.

Definition at line 118 of file readerwriter.h.

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

- /home/moslevin/projects/m3-repo/kernel/src/public/readerwriter.h
- /home/moslevin/projects/m3-repo/kernel/src/readerwriter.cpp

19.30 Mark3::Scheduler Class Reference

The Scheduler Class. This class provides priority-based round-robin Thread scheduling for all active threads managed by the kernel.

`#include <scheduler.h>`

Static Public Member Functions

- static void Init ()

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

  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.
- static void Add (Thread *pclThread_)

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

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

  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.
- static Thread * GetCurrentThread ()

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

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

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

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

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

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

Static Private Attributes

- static constexpr auto m_uNumPriorities = size_t { KERNEL_NUM_PRIORITIES }
- 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 [m_uNumPriorities]

  ThreadLists for all threads at all priorities.
- static PriorityMap m_clPrioMap

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

19.30.1 Detailed Description

The Scheduler Class. This class provides priority-based round-robin Thread scheduling for all active threads managed by the kernel.

Definition at line 63 of file scheduler.h.

19.30.2 Member Function Documentation

19.30.2.1 Add()

```
void Mark3::Scheduler::Add (
            Thread * pclThread_ )  [static]
```

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

Parameters

*pcl↩ Thread_*	Pointer to the thread to add to the scheduler

Definition at line 59 of file scheduler.cpp.

19.30.2.2 GetCurrentThread()

```
static Thread* Mark3::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 116 of file scheduler.h.

19.30.2.3 GetNextThread()

```
static volatile Thread* Mark3::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 124 of file scheduler.h.

19.30.2.4 GetStopList()

```
static ThreadList* Mark3::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 142 of file scheduler.h.

19.30.2.5 GetThreadList()

```
static ThreadList* Mark3::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

*uX←↩ Priority_*	Priority level of the threadlist

Returns

Pointer to the ThreadList for the given priority level

Definition at line 134 of file scheduler.h.

19.30.2.6 Init()

```
void Mark3::Scheduler::Init (
            void )  [static]
```

Init Intiailize the scheduler, must be called before use.

Definition at line 36 of file scheduler.cpp.

19.30.2.7 IsEnabled()

```
static bool Mark3::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 150 of file scheduler.h.

19.30.2.8 QueueScheduler()

```
static void Mark3::Scheduler::QueueScheduler ( ) [inline], [static]
```

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

Definition at line 156 of file scheduler.h.

19.30.2.9 Remove()

```
void Mark3::Scheduler::Remove (
            Thread * pclThread_ ) [static]
```

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

Parameters

*pcl↩ Thread_*	Pointer to the thread to be removed from the scheduler

Definition at line 67 of file scheduler.cpp.

19.30.2.10 Schedule()

```
void Mark3::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 45 of file scheduler.cpp.

19.30.2.11 SetScheduler()

```
bool Mark3::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 75 of file scheduler.cpp.

19.30.3 Member Data Documentation

19.30.3.1 m_aclPriorities

```
ThreadList Mark3::Scheduler::m_aclPriorities [static], [private]
```

ThreadLists for all threads at all priorities.

Definition at line 171 of file scheduler.h.

19.30.3.2 m_bEnabled

```
bool Mark3::Scheduler::m_bEnabled [static], [private]
```

Scheduler's state - enabled or disabled.

Definition at line 162 of file scheduler.h.

19.30.3.3 m_bQueuedSchedule

```
bool Mark3::Scheduler::m_bQueuedSchedule [static], [private]
```

Variable representing whether or not there's a queued scheduler operation.

Definition at line 165 of file scheduler.h.

19.30.3.4 m_clPrioMap

`PriorityMap Mark3::Scheduler::m_clPrioMap [static], [private]`

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

Definition at line 174 of file scheduler.h.

19.30.3.5 m_clStopList

`ThreadList Mark3::Scheduler::m_clStopList [static], [private]`

ThreadList for all stopped threads.

Definition at line 168 of file scheduler.h.

19.30.3.6 m_uNumPriorities

`constexpr auto Mark3::Scheduler::m_uNumPriorities = size_t { KERNEL_NUM_PRIORITIES } [static], [private]`

Definition at line 159 of file scheduler.h.

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

- /home/moslevin/projects/m3-repo/kernel/src/public/scheduler.h
- /home/moslevin/projects/m3-repo/kernel/src/scheduler.cpp

19.31 Mark3::SchedulerGuard Class Reference

The SchedulerGuard class This class implements RAII-based control of the scheduler's global state. Upon object construction, the scheduler's state is cached locally and the scheduler is disabled (if not already disabled). Upon object destruction, the scheduler's previous state is restored. This object is interrupt-safe, although it has no effect when called from an interrupt given that interrupts are inherently higher-priority than threads.

```
#include <schedulerguard.h>
```

Public Member Functions

- SchedulerGuard ()
- ~SchedulerGuard ()

Private Attributes

- bool m_bSchedState

19.31.1 Detailed Description

The SchedulerGuard class This class implements RAII-based control of the scheduler's global state. Upon object construction, the scheduler's state is cached locally and the scheduler is disabled (if not already disabled). Upon object destruction, the scheduler's previous state is restored. This object is interrupt-safe, although it has no effect when called from an interrupt given that interrupts are inherently higher-priority than threads.

Definition at line 37 of file schedulerguard.h.

19.31.2 Constructor & Destructor Documentation

19.31.2.1 SchedulerGuard()

```
Mark3::SchedulerGuard::SchedulerGuard ( )  [inline]
```

Definition at line 39 of file schedulerguard.h.

19.31.2.2 ∼SchedulerGuard()

```
Mark3::SchedulerGuard::∼SchedulerGuard ( )  [inline]
```

Definition at line 44 of file schedulerguard.h.

19.31.3 Member Data Documentation

19.31.3.1 m_bSchedState

```
bool Mark3::SchedulerGuard::m_bSchedState  [private]
```

Definition at line 50 of file schedulerguard.h.

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

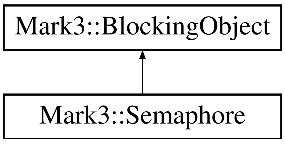
- /home/moslevin/projects/m3-repo/kernel/src/public/schedulerguard.h

19.32 Mark3::Semaphore Class Reference

the Semaphore class provides Binary & Counting semaphore objects, based on BlockingObject base class.

```
#include <ksemaphore.h>
```

Inheritance diagram for Mark3::Semaphore:



Public Member Functions

- void ∗ operator new (size_t sz, void ∗pv)
- ∼Semaphore ()
- void Init (uint16_t u16InitVal_, uint16_t u16MaxVal_)

  Initialize a semaphore before use. Must be called before attempting post/pend operations on the object.
- bool 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.
- void 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.
- uint16_t 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.
- bool 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.
- void 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.

Private Member Functions

- uint8_t WakeNext ()

  Wake the next thread waiting on the semaphore. Used internally.
- 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

19.32.1 Detailed Description

the Semaphore class provides Binary & Counting semaphore objects, based on BlockingObject base class.

Examples:

lab4_semaphores/main.cpp, lab6_timers/main.cpp, and lab9_dynamic_threads/main.cpp.

Definition at line 36 of file ksemaphore.h.

19.32.2 Constructor & Destructor Documentation

19.32.2.1 ∼Semaphore()

```
Mark3::Semaphore::∼Semaphore ( )
```

Definition at line 58 of file ksemaphore.cpp.

19.32.3 Member Function Documentation

19.32.3.1 GetCount()

```
uint16_t Mark3::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 206 of file ksemaphore.cpp.

19.32.3.2 Init()

```
void Mark3::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
*u16Max←Val_*	Maximum value for the semaphore. Must be nonzero.

Examples:

lab6_timers/main.cpp, and lab9_dynamic_threads/main.cpp.

Definition at line 94 of file ksemaphore.cpp.

19.32.3.3 operator new()

```
void* Mark3::Semaphore::operator new (
             size_t sz,
             void * pv )   [inline]
```

Definition at line 39 of file ksemaphore.h.

19.32.3.4 Pend() [1/2]

```
void Mark3::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:

lab4_semaphores/main.cpp.

Definition at line 194 of file ksemaphore.cpp.

19.32.3.5 Pend() [2/2]

```
bool Mark3::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 200 of file ksemaphore.cpp.

19.32.3.6 Pend_i()

```
bool Mark3::Semaphore::Pend_i (
            uint32_t u32WaitTimeMS_ )  [private]
```

Pend_i.

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

Parameters

*u32WaitTimeM↩ S_*	Time in MS to wait

Returns

true on success, false on failure.

Definition at line 152 of file ksemaphore.cpp.

19.32.3.7 Post()

```
bool Mark3::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:

lab4_semaphores/main.cpp, and lab6_timers/main.cpp.

Definition at line 108 of file ksemaphore.cpp.

19.32.3.8 WakeMe()

```
void Mark3::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 68 of file ksemaphore.cpp.

19.32.3.9 WakeNext()

```
uint8_t Mark3::Semaphore::WakeNext ( ) [private]
```

Wake the next thread waiting on the semaphore. Used internally.

Definition at line 78 of file ksemaphore.cpp.

19.32.4 Member Data Documentation

19.32.4.1 m_u16MaxValue

```
uint16_t Mark3::Semaphore::m_u16MaxValue [private]
```

Maximum count that can be held by this semaphore.

Definition at line 140 of file ksemaphore.h.

19.32.4.2 m_u16Value

```
uint16_t Mark3::Semaphore::m_u16Value [private]
```

Current count held by the semaphore.

Definition at line 139 of file ksemaphore.h.

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

- /home/moslevin/projects/m3-repo/kernel/src/public/ksemaphore.h
- /home/moslevin/projects/m3-repo/kernel/src/ksemaphore.cpp

19.33 Mark3::Streamer Class Reference

The Streamer class. This class implements a circular byte-buffer with thread and interrupt safe methods for writing-to and reading-from the buffer. Objects of this class type are designed to be shared between threads, or between threads and interrupts.

```
#include <streamer.h>
```

Public Member Functions

- void Init (uint8_t ∗pau8Buffer_, uint16_t u16Size_)

  Init. Initialize the Streamer object prior to its use, providing a blob of memory for the object to manage.
- bool Read (uint8_t ∗pu8Data_)

  Read. Read a byte of data from the stream, if available.
- uint16_t Read (uint8_t ∗pu8Data_, uint16_t u16Len_)

  Read. Read from the stream until a specified number of bytes have been read, or the stream is exhausted.
- bool Write (uint8_t u8Data_)

  Write. Write a byte of data into the stream.
- uint16_t Write (uint8_t ∗pu8Data_, uint16_t u16Len_)

  Write. Write a maximum number of bytes to the stream.
- bool Claim (uint8_t ∗∗pu8Addr_)

  Claim. Claim a byte of data for writing, without actually writing into it. When the writer is ready to write into the data byte as a result of another operation, it can then populate the byte.
- void Lock (uint8_t ∗pu8LockAddr_)

  Lock. When the lock is set, a client can neither read from, or write to the buffer at the index specified. This is used to in conjunction with Claim to safely reserve data from the buffer, while preventing race conditions occurring as a result of a consumer acting on the data before it is ready.
- void Unlock (void)

  Unlock. Reset the lock pointer in the object, allowing a consumer to read any previously unavailable data that might still be in the stream.
- uint16_t GetAvailable (void)

  GetAvailable.
- bool CanRead (void)

  CanRead.
- bool CanWrite (void)

  CanWrite.
- bool IsEmpty (void)

  IsEmpty.

Private Attributes

- uint8_t ∗ m_pau8Buffer

  Pointer to the buffer managed in this object.
- uint8_t ∗ m_pu8LockAddr

  Address of the lock point in the stream.
- uint16_t m_u16Size

  Size of the stream's circular buffer (in bytes)
- uint16_t m_u16Avail

  Number of bytes free in the stream.
- uint16_t m_u16Head

  Current head index (write to) of the stream.
- uint16_t m_u16Tail

  Current tail index (read from) of the stream.

19.33.1 Detailed Description

The Streamer class. This class implements a circular byte-buffer with thread and interrupt safe methods for writing-to and reading-from the buffer. Objects of this class type are designed to be shared between threads, or between threads and interrupts.

Definition at line 35 of file streamer.h.

19.33.2 Member Function Documentation

19.33.2.1 CanRead()

```
bool Mark3::Streamer::CanRead (
            void )
```

CanRead.

Returns

true if the stream has any unread data

Definition at line 200 of file streamer.cpp.

19.33.2.2 CanWrite()

```
bool Mark3::Streamer::CanWrite (
            void )
```

CanWrite.

Returns

true if the stream has any free space

Definition at line 211 of file streamer.cpp.

19.33.2.3 Claim()

```
bool Mark3::Streamer::Claim (
            uint8_t ** pu8Addr_ )
```

Claim. Claim a byte of data for writing, without actually writing into it. When the writer is ready to write into the data byte as a result of another operation, it can then populate the byte.

This method is useful when encoding data from a raw format into a packet-based format, where one byte of input may result in multiple bytes of output being generated. Especially in cases where a user wants to write to a stream while a peripheral reads from it to transmisit its contents asynchronously (i.e. an interrupt-driven UART consuming a data packet, where the packet is being framed and consumed simultaneously).

This should be used in conjunction with the Lock method and judicious use of critical sections to prevent race conditions.

Parameters

*pu8↩ Addr_*	[out] Pointer to a byte pointer that will contain the address of the "claimed" byte on success.

Returns

true if successful, false if buffer full or locked.

Definition at line 224 of file streamer.cpp.

19.33.2.4 GetAvailable()

```
uint16_t Mark3::Streamer::GetAvailable (
            void ) [inline]
```

GetAvailable.

Returns

The current number of bytes available for write in the streams

Definition at line 134 of file streamer.h.

19.33.2.5 Init()

```
void Mark3::Streamer::Init (
            uint8_t * pau8Buffer_,
            uint16_t u16Size_ )
```

Init. Initialize the Streamer object prior to its use, providing a blob of memory for the object to manage.

Parameters

*pau8↩ Buffer_*	Blob of memory to use as a circular buffer
*u16Size_*	Size of the supplied buffer in bytes

Definition at line 27 of file streamer.cpp.

19.33.2.6 IsEmpty()

```
bool Mark3::Streamer::IsEmpty (
            void )
```

IsEmpty.

Returns

true if the stream is empty

Definition at line 264 of file streamer.cpp.

19.33.2.7 Lock()

```
void Mark3::Streamer::Lock (
            uint8_t * pu8LockAddr_ )
```

Lock. When the lock is set, a client can neither read from, or write to the buffer at the index specified. This is used to in conjunction with Claim to safely reserve data from the buffer, while preventing race conditions occurring as a result of a consumer acting on the data before it is ready.

Parameters

*pu8LockAddr_*	Address (within the stream) to set as the lockpoint.

Definition at line 250 of file streamer.cpp.

19.33.2.8 Read() [1/2]

```
bool Mark3::Streamer::Read (
            uint8_t * pu8Data_ )
```

Read. Read a byte of data from the stream, if available.

Parameters

*pu8Data_*	Pointer to read data into from the stream

Returns

true if data was read, false if data unavailable or buffer locked

Definition at line 38 of file streamer.cpp.

19.33.2.9 Read() [2/2]

```
uint16_t Mark3::Streamer::Read (
            uint8_t * pu8Data_,
            uint16_t u16Len_ )
```

Read. Read from the stream until a specified number of bytes have been read, or the stream is exhausted.

Parameters

*pu8Data_*	pointer to an array of data read into
*u16Len_*	maximum number of bytes to read

Returns

number of bytes read

Definition at line 64 of file streamer.cpp.

19.33.2.10 Unlock()

```
void Mark3::Streamer::Unlock (
            void )
```

Unlock. Reset the lock pointer in the object, allowing a consumer to read any previously unavailable data that might still be in the stream.

Definition at line 257 of file streamer.cpp.

19.33.2.11 Write() [1/2]

```
bool Mark3::Streamer::Write (
            uint8_t u8Data_ )
```

Write. Write a byte of data into the stream.

Parameters

*u8Data_*	Data byte to be written into the stream

Returns

true if byte was written, false on buffer full or buffer locked at index.

Definition at line 120 of file streamer.cpp.

19.33.2.12 Write() [2/2]

```
uint16_t Mark3::Streamer::Write (
            uint8_t * pu8Data_,
            uint16_t u16Len_ )
```

Write. Write a maximum number of bytes to the stream.

Parameters

*pu8←Data_*	pointer to an array of bytes to write out to the stream
u16Len←_	Length of data held in the array

Returns

number of bytes written to the stream

Definition at line 144 of file streamer.cpp.

19.33.3 Member Data Documentation

19.33.3.1 m_pau8Buffer

```
uint8_t* Mark3::Streamer::m_pau8Buffer [private]
```

Pointer to the buffer managed in this object.

Definition at line 154 of file streamer.h.

19.33.3.2 m_pu8LockAddr

```
uint8_t* Mark3::Streamer::m_pu8LockAddr [private]
```

Address of the lock point in the stream.

Definition at line 155 of file streamer.h.

19.33.3.3 m_u16Avail

```
uint16_t Mark3::Streamer::m_u16Avail [private]
```

Number of bytes free in the stream.

Definition at line 158 of file streamer.h.

19.33.3.4 m_u16Head

```
uint16_t Mark3::Streamer::m_u16Head [private]
```

Current head index (write to) of the stream.

Definition at line 159 of file streamer.h.

19.33.3.5 m_u16Size

```
uint16_t Mark3::Streamer::m_u16Size [private]
```

Size of the stream's circular buffer (in bytes)

Definition at line 157 of file streamer.h.

19.33.3.6 m_u16Tail

```
uint16_t Mark3::Streamer::m_u16Tail [private]
```

Current tail index (read from) of the stream.

Definition at line 160 of file streamer.h.

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

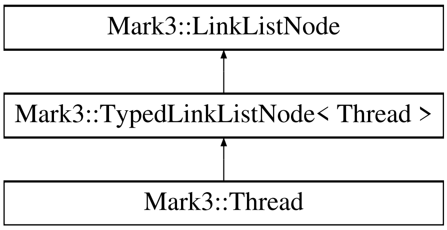
- /home/moslevin/projects/m3-repo/kernel/lib/streamer/public/streamer.h
- /home/moslevin/projects/m3-repo/kernel/lib/streamer/streamer.cpp

19.34 Mark3::Thread Class Reference

The Thread Class. This object providing the fundamental thread control data structures and functions that define a single thread of execution in the Mark3 operating system. It is the fundamental data type used to provide multitasking support in the kernel.

```
#include <thread.h>
```

Inheritance diagram for Mark3::Thread:



Public Member Functions

- void ∗ operator new (size_t sz, void ∗pv)
- ∼Thread ()
- Thread ()
- bool IsInitialized ()

  IsInitialized Used to check whether or not a thread has been initialized prior to use.
- void Init (K_WORD ∗pwStack_, uint16_t u16StackSize_, PORT_PRIO_TYPE uXPriority_, ThreadEntryFunc 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.
- void Start ()

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

  Stop Stop a thread that's actively scheduled without destroying its stacks. Stopped threads can be restarted using the Start() API.
- void SetName (const char ∗szName_)

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

  GetName.
- ThreadList ∗ GetOwner (void)

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

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

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

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

  SetQuantum Set the thread's round-robin execution quantum.
- uint16_t GetQuantum (void)

  GetQuantum Get the thread's round-robin execution quantum.

- void SetCurrent (ThreadList ∗pclNewList_)

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

  SetOwner. Set the thread's owner to the specified thread list.
- void 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.
- void 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.
- void 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.
- void SetID (uint8_t u8ID_)

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

  GetID Return the thread's integer ID. Note that this ID is not guaranteed to be unique when dynamic threading is used in the system, or there are more than 255 threads. Also not guaranteed to be unique if the SetID function is called by the user.
- uint16_t 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.
- 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 eMode_)

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

  GetEventFlagMode Returns the thread's event flag's operating mode.
- Timer ∗ GetTimer ()
- void SetExpired (bool bExpired_)

  SetExpired Set the status of the current blocking call on the thread.
- bool GetExpired ()

  GetExpired Return the status of the most-recent blocking call on the thread.
- void ∗ GetExtendedContext ()

  GetExtendedContext Return the Thread object's extended-context data pointer. Used by code implementing a user-defined thread-local storage model. Pointer exists only for the lifespan of the Thread.
- void SetExtendedContext (void ∗pvData_)

  SetExtendedContext Assign the Thread object's extended-context data pointer. Used by code implementing a user-defined thread-local storage model.
- ThreadState GetState ()

  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.
- void SetState (ThreadState eState_)

  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.
- K_WORD ∗ GetStack ()

  GetStack.
- uint16_t GetStackSize ()

  GetStackSize.

Static Public Member Functions

- static Thread ∗ Init (uint16_t u16StackSize_, PORT_PRIO_TYPE uXPriority_, ThreadEntryFunc pfEntry←Point_, void ∗pvArg_)

  Init Create and initialize a new thread, using memory from the auto-allocated heap region to supply both the thread object and its stack. The thread returned can then be started using the Start() method directly. Note that the memory used to create this thread cannot be reclaimed, and so this API is only suitable for threads that exist for the duration of runtime.
- static void Sleep (uint32_t u32TimeMs_)

  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.
- static void Yield (void)

  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.
- static void CoopYield (void)

  CoopYield Cooperative yield - This forces the system to not only call the scheduler, but also move the currently executing thread to the back of the current thread list, allowing other same-priority threads the opportunity to run. This is used primarily for cooperative scheduling between threads in the same priority level.

Private Member Functions

- void SetPriorityBase (PORT_PRIO_TYPE uXPriority_)

  SetPriorityBase.

Static Private Member Functions

- static void ContextSwitchSWI (void)

  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.

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 m_eState

  Enum indicating the thread's current state.
- void ∗ m_pvExtendedContext

  Pointer provided to a Thread to implement thread-local storage.
- const char ∗ m_szName

  Thread name.
- 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.
- ThreadEntryFunc 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 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

19.34.1 Detailed Description

The Thread Class. This object providing the fundamental thread control data structures and functions that define a single thread of execution in the Mark3 operating system. It is the fundamental data type used to provide multitasking support in the kernel.

Examples:

lab10_notifications/main.cpp, lab11_mailboxes/main.cpp, lab1_kernel_setup/main.cpp, lab2_coroutines/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 64 of file thread.h.

19.34.2 Constructor & Destructor Documentation

19.34.2.1 ∼Thread()

```
Mark3::Thread::~Thread ( )
```

Definition at line 30 of file thread.cpp.

19.34.2.2 Thread()

```
Mark3::Thread::Thread ( ) [inline]
```

Definition at line 70 of file thread.h.

19.34.3 Member Function Documentation

19.34.3.1 ContextSwitchSWI()

```
void Mark3::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 392 of file thread.cpp.

19.34.3.2 CoopYield()

```
void Mark3::Thread::CoopYield (
            void ) [static]
```

CoopYield Cooperative yield - This forces the system to not only call the scheduler, but also move the currently executing thread to the back of the current thread list, allowing other same-priority threads the opportunity to run. This is used primarily for cooperative scheduling between threads in the same priority level.

Definition at line 325 of file thread.cpp.

19.34.3.3 Exit()

```
void Mark3::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 187 of file thread.cpp.

19.34.3.4 GetCurPriority()

```
PORT_PRIO_TYPE Mark3::Thread::GetCurPriority (
            void ) [inline]
```

GetCurPriority Return the priority of the current thread.

Returns

Priority of the current thread

Definition at line 181 of file thread.h.

19.34.3.5 GetCurrent()

```
ThreadList* Mark3::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 166 of file thread.h.

19.34.3.6 GetEventFlagMask()

```
uint16_t Mark3::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 317 of file thread.h.

19.34.3.7 GetEventFlagMode()

```
EventFlagOperation Mark3::Thread::GetEventFlagMode ( ) [inline]
```

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

Returns

The thread's event flag mode.

Definition at line 336 of file thread.h.

19.34.3.8 GetExpired()

```
bool Mark3::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 425 of file thread.cpp.

19.34.3.9 GetExtendedContext()

```
void* Mark3::Thread::GetExtendedContext ( ) [inline]
```

GetExtendedContext Return the Thread object's extended-context data pointer. Used by code implementing a user-defined thread-local storage model. Pointer exists only for the lifespan of the Thread.

Returns

Thread's extended context data pointer.

Definition at line 368 of file thread.h.

19.34.3.10 GetID()

```
uint8_t Mark3::Thread::GetID ( ) [inline]
```

GetID Return the thread's integer ID. Note that this ID is not guaranteed to be unique when dynamic threading is used in the system, or there are more than 255 threads. Also not guaranteed to be unique if the SetID function is called by the user.

Returns

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

Definition at line 292 of file thread.h.

19.34.3.11 GetName()

```
const char* Mark3::Thread::GetName ( ) [inline]
```

GetName.

Returns

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

Definition at line 149 of file thread.h.

19.34.3.12 GetOwner()

```
ThreadList* Mark3::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 159 of file thread.h.

19.34.3.13 GetPriority()

```
PORT_PRIO_TYPE Mark3::Thread::GetPriority (
            void ) [inline]
```

GetPriority Return the priority of the current thread.

Returns

Priority of the current thread

Definition at line 174 of file thread.h.

19.34.3.14 GetQuantum()

```
uint16_t Mark3::Thread::GetQuantum (
             void ) [inline]
```

GetQuantum Get the thread's round-robin execution quantum.

Returns

The thread's quantum

Definition at line 197 of file thread.h.

19.34.3.15 GetStack()

```
K_WORD* Mark3::Thread::GetStack ( ) [inline]
```

GetStack.

Returns

Pointer to the blob of memory used as the thread's stack

Definition at line 403 of file thread.h.

19.34.3.16 GetStackSize()

```
uint16_t Mark3::Thread::GetStackSize ( ) [inline]
```

GetStackSize.

Returns

Size of the thread's stack in bytes

Definition at line 409 of file thread.h.

19.34.3.17 GetStackSlack()

```
uint16_t Mark3::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

Examples:

lab9_dynamic_threads/main.cpp.

19.34.3.18 GetState()

```
ThreadState Mark3::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 389 of file thread.h.

19.34.3.19 GetTimer()

```
Timer * Mark3::Thread::GetTimer ( )
```

Return a pointer to the thread's timer object

Definition at line 412 of file thread.cpp.

19.34.3.20 InheritPriority()

```
void Mark3::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

uX↩ *Priority_*	New Priority to boost to.

Definition at line 383 of file thread.cpp.

19.34.3.21 Init() [1/2]

```
void Mark3::Thread::Init (
            K_WORD * pwStack_,
            uint16_t u16StackSize_,
            PORT_PRIO_TYPE uXPriority_,
            ThreadEntryFunc 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
u16Stack↩ *Size_*	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:

lab9_dynamic_threads/main.cpp.

Definition at line 49 of file thread.cpp.

19.34.3.22 Init() [2/2]

```
Thread * Mark3::Thread::Init (
            uint16_t u16StackSize_,
            PORT_PRIO_TYPE uXPriority_,
            ThreadEntryFunc pfEntryPoint_,
            void * pvArg_ ) [static]
```

Init Create and initialize a new thread, using memory from the auto-allocated heap region to supply both the thread object and its stack. The thread returned can then be started using the Start() method directly. Note that the memory used to create this thread cannot be reclaimed, and so this API is only suitable for threads that exist for the duration of runtime.

Parameters

*u16Stack←↩ Size_*	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.

Returns

Pointer to a newly-created thread.

Definition at line 101 of file thread.cpp.

19.34.3.23 IsInitialized()

```
bool Mark3::Thread::IsInitialized (
             void  )  [inline]
```

IsInitialized Used to check whether or not a thread has been initialized prior to use.

Returns

return true if the thread object has been initialized, false otherwise.

Definition at line 77 of file thread.h.

19.34.3.24 operator new()

```
void* Mark3::Thread::operator new (
             size_t sz,
             void * pv )  [inline]
```

Definition at line 67 of file thread.h.

19.34.3.25 SetCurrent()

```
void Mark3::Thread::SetCurrent (
             ThreadList * pclNewList_ )  [inline]
```

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

Parameters

*pclNew↩ List_*	Pointer to the threadlist to apply thread ownership

Definition at line 206 of file thread.h.

19.34.3.26 SetEventFlagMask()

```
void Mark3::Thread::SetEventFlagMask (
            uint16_t u16Mask_ )  [inline]
```

SetEventFlagMask Sets the active event flag bitfield mask.

Parameters

*u16↩ Mask_*	

Definition at line 323 of file thread.h.

19.34.3.27 SetEventFlagMode()

```
void Mark3::Thread::SetEventFlagMode (
            EventFlagOperation eMode_ )  [inline]
```

SetEventFlagMode Sets the active event flag operation mode.

Parameters

e↩ Mode↩ _	Event flag operation mode, defines the logical operator to apply to the event flag.

Definition at line 330 of file thread.h.

19.34.3.28 SetExpired()

```
void Mark3::Thread::SetExpired (
            bool bExpired_ )
```

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

Parameters

b↩ Expired↩ _	true - call expired, false - call did not expire

Definition at line 418 of file thread.cpp.

19.34.3.29 SetExtendedContext()

```
void Mark3::Thread::SetExtendedContext (
            void * pvData_ )  [inline]
```

SetExtendedContext Assign the Thread object's extended-context data pointer. Used by code implementing a user-defined thread-local storage model.

Object assigned to the context pointer should persist for the duration of the Thread.

Parameters

*pv↩ Data_*	Object to assign to the extended data pointer.+

Definition at line 380 of file thread.h.

19.34.3.30 SetID()

```
void Mark3::Thread::SetID (
            uint8_t u8ID_ )  [inline]
```

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

Parameters

*u8I↩ D_*	8-bit Thread ID, set by the user

Definition at line 282 of file thread.h.

19.34.3.31 SetName()

```
void Mark3::Thread::SetName (
            const char * szName_ )  [inline]
```

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

Parameters

sz← *Name_*	Char string containing the thread name

Definition at line 143 of file thread.h.

19.34.3.32 SetOwner()

```
void Mark3::Thread::SetOwner (
            ThreadList * pclNewList_ )  [inline]
```

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

Parameters

pclNew← *List_*	Pointer to the threadlist to apply thread ownership

Definition at line 213 of file thread.h.

19.34.3.33 SetPriority()

```
void Mark3::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

uX← *Priority_*	New priority of the thread

Definition at line 342 of file thread.cpp.

19.34.3.34 SetPriorityBase()

```
void Mark3::Thread::SetPriorityBase (
            PORT_PRIO_TYPE uXPriority_ )  [private]
```

SetPriorityBase.

Parameters

*uX↩ Priority_*	

Definition at line 332 of file thread.cpp.

19.34.3.35 SetQuantum()

```
void Mark3::Thread::SetQuantum (
            uint16_t u16Quantum_ )  [inline]
```

SetQuantum Set the thread's round-robin execution quantum.

Parameters

*u16↩ Quantum_*	Thread's execution quantum (in milliseconds)

Definition at line 190 of file thread.h.

19.34.3.36 SetState()

```
void Mark3::Thread::SetState (
            ThreadState 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

e↩ State↩ _	New thread state to set.

Definition at line 397 of file thread.h.

19.34.3.37 Sleep()

```
void Mark3::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

*u32Time↩ Ms_*	Time to sleep (in ms)

Examples:

lab10_notifications/main.cpp, lab11_mailboxes/main.cpp, lab1_kernel_setup/main.cpp, lab2_coroutines/main.↩ cpp, lab7_events/main.cpp, lab8_messages/main.cpp, and lab9_dynamic_threads/main.cpp.

Definition at line 249 of file thread.cpp.

19.34.3.38 Start()

```
void Mark3::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:

lab9_dynamic_threads/main.cpp.

Definition at line 110 of file thread.cpp.

19.34.3.39 Stop()

```
void Mark3::Thread::Stop (
             void  )
```

Stop Stop a thread that's actively scheduled without destroying its stacks. Stopped threads can be restarted using the Start() API.

Definition at line 141 of file thread.cpp.

19.34.3.40 Yield()

```
void Mark3::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 305 of file thread.cpp.

19.34.4 Friends And Related Function Documentation

19.34.4.1 ThreadPort

```
friend class ThreadPort [friend]
```

Definition at line 411 of file thread.h.

19.34.5 Member Data Documentation

19.34.5.1 m_bExpired

```
bool Mark3::Thread::m_bExpired [private]
```

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

Definition at line 489 of file thread.h.

19.34.5.2 m_clTimer

```
Timer Mark3::Thread::m_clTimer [private]
```

Timer used for blocking-object timeouts.

Definition at line 486 of file thread.h.

19.34.5.3 m_eFlagMode

```
EventFlagOperation Mark3::Thread::m_eFlagMode [private]
```

Event-flag mode.

Definition at line 482 of file thread.h.

19.34.5.4 m_eState

`ThreadState Mark3::Thread::m_eState [private]`

Enum indicating the thread's current state.

Definition at line 445 of file thread.h.

19.34.5.5 m_pclCurrent

`ThreadList* Mark3::Thread::m_pclCurrent [private]`

Pointer to the thread-list where the thread currently resides.

Definition at line 461 of file thread.h.

19.34.5.6 m_pclOwner

`ThreadList* Mark3::Thread::m_pclOwner [private]`

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

Definition at line 464 of file thread.h.

19.34.5.7 m_pfEntryPoint

`ThreadEntryFunc Mark3::Thread::m_pfEntryPoint [private]`

The entry-point function called when the thread starts.

Definition at line 467 of file thread.h.

19.34.5.8 m_pvArg

`void* Mark3::Thread::m_pvArg [private]`

Pointer to the argument passed into the thread's entrypoint.

Definition at line 470 of file thread.h.

19.34.5.9 m_pvExtendedContext

```
void* Mark3::Thread::m_pvExtendedContext [private]
```

Pointer provided to a Thread to implement thread-local storage.

Definition at line 449 of file thread.h.

19.34.5.10 m_pwStack

```
K_WORD* Mark3::Thread::m_pwStack [private]
```

Pointer to the thread's stack.

Definition at line 433 of file thread.h.

19.34.5.11 m_pwStackTop

```
K_WORD* Mark3::Thread::m_pwStackTop [private]
```

Pointer to the top of the thread's stack.

Definition at line 430 of file thread.h.

19.34.5.12 m_szName

```
const char* Mark3::Thread::m_szName [private]
```

Thread name.

Definition at line 454 of file thread.h.

19.34.5.13 m_u16FlagMask

```
uint16_t Mark3::Thread::m_u16FlagMask [private]
```

Event-flag mask.

Definition at line 479 of file thread.h.

19.34.5.14 m_u16Quantum

`uint16_t Mark3::Thread::m_u16Quantum [private]`

Thread quantum (in milliseconds)

Definition at line 474 of file thread.h.

19.34.5.15 m_u16StackSize

`uint16_t Mark3::Thread::m_u16StackSize [private]`

Size of the stack (in bytes)

Definition at line 458 of file thread.h.

19.34.5.16 m_u8ThreadID

`uint8_t Mark3::Thread::m_u8ThreadID [private]`

Thread ID.

Definition at line 436 of file thread.h.

19.34.5.17 m_uXCurPriority

`PORT_PRIO_TYPE Mark3::Thread::m_uXCurPriority [private]`

Current priority of the thread (priority inheritence)

Definition at line 442 of file thread.h.

19.34.5.18 m_uXPriority

`PORT_PRIO_TYPE Mark3::Thread::m_uXPriority [private]`

Default priority of the thread.

Definition at line 439 of file thread.h.

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

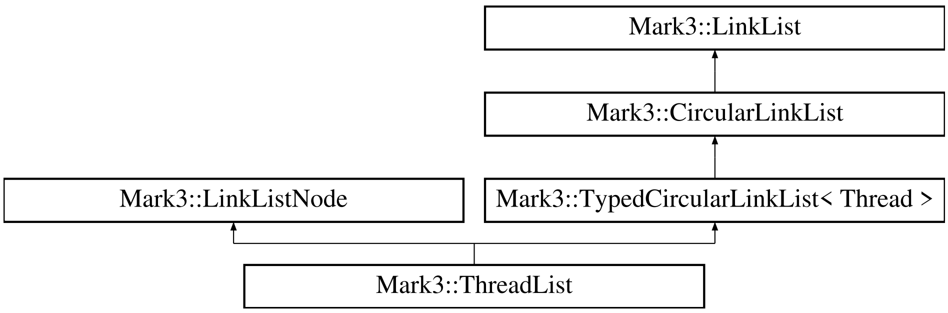
- /home/moslevin/projects/m3-repo/kernel/src/public/thread.h
- /home/moslevin/projects/m3-repo/kernel/src/thread.cpp

19.35 Mark3::ThreadList Class Reference

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

```
#include <threadlist.h>
```

Inheritance diagram for Mark3::ThreadList:



Public Member Functions

- void ∗ operator new (size_t sz, void ∗pv)
- ThreadList ()

  ThreadList Default constructor - zero-initializes the data.
- void SetPriority (PORT_PRIO_TYPE uXPriority_)

  SetPriority Set the priority of this threadlist (if used for a scheduler).
- void 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.
- void Add (Thread ∗node_)

  Add Add a thread to the threadlist.
- void Add (Thread ∗node_, PriorityMap ∗pclMap_, PORT_PRIO_TYPE uXPriority_)

  Add Add a thread to the threadlist, specifying the flag and priority at the same time.
- void AddPriority (Thread ∗node_)

  AddPriority Add a thread to the list such that threads are ordered from highest to lowest priority from the head of the list.
- void Remove (Thread ∗node_)

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

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

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

19.35.1 Detailed Description

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

Definition at line 38 of file threadlist.h.

19.35.2 Constructor & Destructor Documentation

19.35.2.1 ThreadList()

```
Mark3::ThreadList::ThreadList ( )
```

ThreadList Default constructor - zero-initializes the data.

Definition at line 27 of file threadlist.cpp.

19.35.3 Member Function Documentation

19.35.3.1 Add() [1/2]

```
void Mark3::ThreadList::Add (
            Thread * 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 47 of file threadlist.cpp.

19.35.3.2 Add() [2/2]

```
void Mark3::ThreadList::Add (
            Thread * 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 nullptr for non-scheduler.
*uX↩ Priority_*	Priority of the threadlist

Definition at line 101 of file threadlist.cpp.

19.35.3.3 AddPriority()

```
void Mark3::ThreadList::AddPriority (
            Thread * 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 66 of file threadlist.cpp.

19.35.3.4 HighestWaiter()

```
Thread * Mark3::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 128 of file threadlist.cpp.

19.35.3.5 operator new()

```
void* Mark3::ThreadList::operator new (
            size_t sz,
            void * pv ) [inline]
```

Definition at line 41 of file threadlist.h.

19.35.3.6 Remove()

```
void Mark3::ThreadList::Remove (
            Thread * 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.

19.35.3.7 SetMapPointer()

```
void Mark3::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

*pcl←↩ Map_*	Pointer to the priority map object used to track this thread.

Definition at line 40 of file threadlist.cpp.

19.35.3.8 SetPriority()

```
void Mark3::ThreadList::SetPriority (
            PORT_PRIO_TYPE uXPriority_ )
```

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

Parameters

*uX←↩ Priority_*	Priority level of the thread list

Definition at line 34 of file threadlist.cpp.

19.35.4 Member Data Documentation

19.35.4.1 m_pclMap

```
PriorityMap* Mark3::ThreadList::m_pclMap [private]
```

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

Definition at line 116 of file threadlist.h.

19.35.4.2 m_uXPriority

`PORT_PRIO_TYPE Mark3::ThreadList::m_uXPriority [private]`

Priority of the threadlist.

Definition at line 113 of file threadlist.h.

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

- /home/moslevin/projects/m3-repo/kernel/src/public/threadlist.h
- /home/moslevin/projects/m3-repo/kernel/src/threadlist.cpp

19.36 Mark3::ThreadListList Class Reference

The ThreadListList class Class used to track all threadlists active in the OS kernel. At any point in time, the list can be traversed to get a complete view of all running, blocked, or stopped threads in the system.

`#include <threadlistlist.h>`

Static Public Member Functions

- static void Add (ThreadList ∗pclThreadList_)

  Add Add a ThreadList to the list for tracking.
- static void Remove (ThreadList ∗pclThreadList_)

  Remove Remove a threadlist from tracking.
- static ThreadList ∗ GetHead ()

  GetHead.

Static Private Attributes

- static TypedDoubleLinkList< ThreadList > m_clThreadListList

19.36.1 Detailed Description

The ThreadListList class Class used to track all threadlists active in the OS kernel. At any point in time, the list can be traversed to get a complete view of all running, blocked, or stopped threads in the system.

Definition at line 36 of file threadlistlist.h.

19.36.2 Member Function Documentation

19.36.2.1 Add()

```
static void Mark3::ThreadListList::Add (
            ThreadList * pclThreadList_ )  [inline], [static]
```

Add Add a ThreadList to the list for tracking.

Parameters

*pclThread↩ List_*	threadlist to add for tracking

Definition at line 43 of file threadlistlist.h.

19.36.2.2 GetHead()

```
static ThreadList* Mark3::ThreadListList::GetHead ( )  [inline], [static]
```

GetHead.

Returns

The threadlist at the beginning of the list

Definition at line 62 of file threadlistlist.h.

19.36.2.3 Remove()

```
static void Mark3::ThreadListList::Remove (
            ThreadList * pclThreadList_ )  [inline], [static]
```

Remove Remove a threadlist from tracking.

Parameters

*pclThread↩ List_*	threadlist to remove from tracking

Definition at line 53 of file threadlistlist.h.

19.36.3 Member Data Documentation

19.36.3.1 m_clThreadListList

```
TypedDoubleLinkList< ThreadList > Mark3::ThreadListList::m_clThreadListList  [static], [private]
```

Definition at line 68 of file threadlistlist.h.

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

- /home/moslevin/projects/m3-repo/kernel/src/public/threadlistlist.h
- /home/moslevin/projects/m3-repo/kernel/src/threadlistlist.cpp

19.37 Mark3::ThreadPort Class Reference

The ThreadPort Class defines the target-specific functions required by the kernel for threading.

`#include <ithreadport.h>`

Static Public Member Functions

- static void Init ()
  Init Function to perform early init of the target environment prior to using OS primatives.
- static void StartThreads ()
  StartThreads Function to start the scheduler, initial threads, etc.

Static Private Member Functions

- static void InitStack (Thread ∗pstThread_)
  InitStack Initialize the thread's stack.

Friends

- class Thread

19.37.1 Detailed Description

The ThreadPort Class defines the target-specific functions required by the kernel for threading.

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 35 of file ithreadport.h.

19.37.2 Member Function Documentation

19.37.2.1 Init()

```
static void Mark3::ThreadPort::Init (
            void  )  [inline], [static]
```

Init Function to perform early init of the target environment prior to using OS primatives.

Definition at line 43 of file ithreadport.h.

19.37.2.2 InitStack()

```
void Mark3::ThreadPort::InitStack (
            Thread * pstThread_ )  [static], [private]
```

InitStack Initialize the thread's stack.

Parameters

*pstThread_*	Pointer to the thread to initialize

Definition at line 43 of file threadport.cpp.

19.37.2.3 StartThreads()

```
void Mark3::ThreadPort::StartThreads ( )  [static]
```

StartThreads Function to start the scheduler, initial threads, etc.

Definition at line 100 of file threadport.cpp.

19.37.3 Friends And Related Function Documentation

19.37.3.1 Thread

```
friend class Thread  [friend]
```

Definition at line 50 of file ithreadport.h.

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

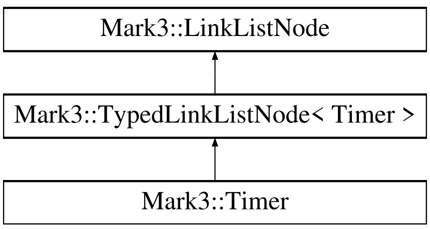
- /home/moslevin/projects/m3-repo/kernel/src/public/ithreadport.h
- /home/moslevin/projects/m3-repo/kernel/src/arch/avr/atmega1284p/gcc/threadport.cpp

19.38 Mark3::Timer Class Reference

The Timer Class. This class provides kernel-managed timers, used to provide high-precision delays. Functionality is useful to both user-code, and is used extensively within the kernel and its blocking objects to implement round-robin scheduling, thread sleep, and timeouts. Provides one-shot and periodic timers for use by application code. This object relies on a target-defined hardware timer implementation, which is multiplexed by the kernel's timer scheduler.

```
#include <timer.h>
```

Inheritance diagram for Mark3::Timer:



Public Member Functions

- void ∗ operator new (size_t sz, void ∗pv)
- ∼Timer ()
- Timer ()

  Timer Default Constructor - Do nothing. Allow the init call to perform the necessary object initialization prior to use.
- void Init ()

  Init Re-initialize the Timer to default values.
- void Start (bool bRepeat_, uint32_t u32IntervalMs_, TimerCallback pfCallback_, void ∗pvData_)

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

  Start Start or restart a timer using parameters previously configured via calls to Start(<with args>), or via the a-la-carte parameter setter methods. This is especially useful for retriggering one-shot timers that have previously expired, using the timer's previous configuration.
- void Stop ()

  Stop Stop a timer already in progress. Has no effect on timers that have already been stopped.

Private Member Functions

- void SetInitialized ()

  SetInitialized.
- bool IsInitialized (void)

  IsInitialized.

Static Private Member Functions

- static uint32_t SecondsToTicks (uint32_t x)
- static uint32_t MSecondsToTicks (uint32_t x)
- static uint32_t USecondsToTicks (uint32_t x)

Private Attributes

- uint8_t m_u8Initialized

  Cookie used to determine whether or not the timer is initialized.
- uint8_t m_u8Flags

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

  Pointer to the callback function.
- uint32_t m_u32Interval

  Interval of the timer in timer ticks.
- uint32_t m_u32TimeLeft

  Time remaining on the timer.
- Thread ∗ m_pclOwner

  Pointer to the owner thread.
- void ∗ m_pvData

  Pointer to the callback data.

Static Private Attributes

- static constexpr auto m_uTimerInvalidCookie = uint8_t { 0x3C }
- static constexpr auto m_uTimerInitCookie = uint8_t { 0xC3 }

Friends

- class TimerList

Additional Inherited Members

19.38.1 Detailed Description

The Timer Class. This class provides kernel-managed timers, used to provide high-precision delays. Functionality is useful to both user-code, and is used extensively within the kernel and its blocking objects to implement round-robin scheduling, thread sleep, and timeouts. Provides one-shot and periodic timers for use by application code. This object relies on a target-defined hardware timer implementation, which is multiplexed by the kernel's timer scheduler.

Examples:

lab6_timers/main.cpp.

Definition at line 68 of file timer.h.

19.38.2 Constructor & Destructor Documentation

19.38.2.1 ∼Timer()

```
Mark3::Timer::∼Timer ( ) [inline]
```

Definition at line 72 of file timer.h.

19.38.2.2 Timer()

```
Mark3::Timer::Timer ( )
```

Timer Default Constructor - Do nothing. Allow the init call to perform the necessary object initialization prior to use.

Definition at line 29 of file timer.cpp.

19.38.3 Member Function Documentation

19.38.3.1 Init()

```
void Mark3::Timer::Init (
             void  )
```

Init Re-initialize the Timer to default values.

Definition at line 36 of file timer.cpp.

19.38.3.2 IsInitialized()

```
bool Mark3::Timer::IsInitialized (
             void  )  [inline], [private]
```

IsInitialized.

Returns

true if initialized, false if not initialized.

Definition at line 127 of file timer.h.

19.38.3.3 MSecondsToTicks()

```
static uint32_t Mark3::Timer::MSecondsToTicks (
             uint32_t x )  [inline], [static], [private]
```

Definition at line 130 of file timer.h.

19.38.3.4 operator new()

```
void* Mark3::Timer::operator new (
             size_t sz,
             void * pv )  [inline]
```

Definition at line 71 of file timer.h.

19.38.3.5 SecondsToTicks()

```
static uint32_t Mark3::Timer::SecondsToTicks (
            uint32_t x ) [inline], [static], [private]
```

Definition at line 129 of file timer.h.

19.38.3.6 SetInitialized()

```
void Mark3::Timer::SetInitialized (
            void ) [inline], [private]
```

SetInitialized.

Definition at line 121 of file timer.h.

19.38.3.7 Start() [1/2]

```
void Mark3::Timer::Start (
            bool bRepeat_,
            uint32_t u32IntervalMs_,
            TimerCallback pfCallback_,
            void * pvData_ )
```

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

Parameters

*bRepeat_*	0 - timer is one-shot. 1 - timer is repeating.
*u32IntervalMs_*	- Interval of the timer in miliseconds
*pfCallback_*	- Function to call on timer expiry
*pvData_*	- Data to pass into the callback function

Examples:

lab6_timers/main.cpp.

Definition at line 51 of file timer.cpp.

19.38.3.8 Start() [2/2]

```
void Mark3::Timer::Start (
            void )
```

Start Start or restart a timer using parameters previously configured via calls to Start(<with args>), or via the a-la-carte parameter setter methods. This is especially useful for retriggering one-shot timers that have previously expired, using the timer's previous configuration.

Definition at line 73 of file timer.cpp.

19.38.3.9 Stop()

```
void Mark3::Timer::Stop (
            void )
```

Stop Stop a timer already in progress. Has no effect on timers that have already been stopped.

Definition at line 86 of file timer.cpp.

19.38.3.10 USecondsToTicks()

```
static uint32_t Mark3::Timer::USecondsToTicks (
            uint32_t x ) [inline], [static], [private]
```

Definition at line 131 of file timer.h.

19.38.4 Friends And Related Function Documentation

19.38.4.1 TimerList

```
friend class TimerList [friend]
```

Definition at line 116 of file timer.h.

19.38.5 Member Data Documentation

19.38.5.1 m_pclOwner

```
Thread* Mark3::Timer::m_pclOwner [private]
```

Pointer to the owner thread.

Definition at line 152 of file timer.h.

19.38.5.2 m_pfCallback

`TimerCallback Mark3::Timer::m_pfCallback [private]`

Pointer to the callback function.

Definition at line 143 of file timer.h.

19.38.5.3 m_pvData

`void* Mark3::Timer::m_pvData [private]`

Pointer to the callback data.

Definition at line 155 of file timer.h.

19.38.5.4 m_u32Interval

`uint32_t Mark3::Timer::m_u32Interval [private]`

Interval of the timer in timer ticks.

Definition at line 146 of file timer.h.

19.38.5.5 m_u32TimeLeft

`uint32_t Mark3::Timer::m_u32TimeLeft [private]`

Time remaining on the timer.

Definition at line 149 of file timer.h.

19.38.5.6 m_u8Flags

`uint8_t Mark3::Timer::m_u8Flags [private]`

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

Definition at line 140 of file timer.h.

19.38.5.7 m_u8Initialized

`uint8_t Mark3::Timer::m_u8Initialized [private]`

Cookie used to determine whether or not the timer is initialized.

Definition at line 137 of file timer.h.

19.38.5.8 m_uTimerInitCookie

`constexpr auto Mark3::Timer::m_uTimerInitCookie = uint8_t { 0xC3 } [static], [private]`

Definition at line 134 of file timer.h.

19.38.5.9 m_uTimerInvalidCookie

`constexpr auto Mark3::Timer::m_uTimerInvalidCookie = uint8_t { 0x3C } [static], [private]`

Definition at line 133 of file timer.h.

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

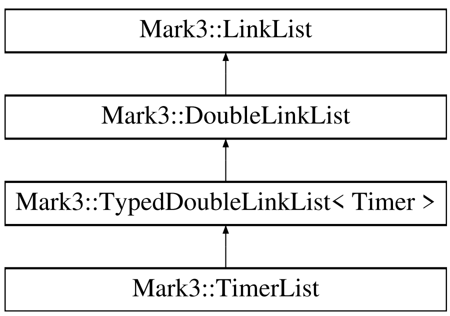
- /home/moslevin/projects/m3-repo/kernel/src/public/timer.h
- /home/moslevin/projects/m3-repo/kernel/src/timer.cpp

19.39 Mark3::TimerList Class Reference

the TimerList class. This class implements a doubly-linked-list of timer objects.

```
#include <timerlist.h>
```

Inheritance diagram for Mark3::TimerList:



Public Member Functions

- void Init ()

    Init Initialize the TimerList object. Must be called before using the object.
- void Add (Timer *pclListNode_)

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

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

    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.

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.
- Mutex m_clMutex

    Guards against concurrent access to the timer list - Only needed when running threaded.

Additional Inherited Members

19.39.1 Detailed Description

the TimerList class. This class implements a doubly-linked-list of timer objects.

Definition at line 40 of file timerlist.h.

19.39.2 Member Function Documentation

19.39.2.1 Add()

```
void Mark3::TimerList::Add (
            Timer * pclListNode_ )
```

Add Add a timer to the TimerList.

Parameters

*pclList←Node_*	Pointer to the Timer to Add

Definition at line 35 of file timerlist.cpp.

19.39.2.2 Init()

```
void Mark3::TimerList::Init (
            void )
```

Init Initialize the TimerList object. Must be called before using the object.

Definition at line 27 of file timerlist.cpp.

19.39.2.3 Process()

```
void Mark3::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.

Definition at line 61 of file timerlist.cpp.

19.39.2.4 Remove()

```
void Mark3::TimerList::Remove (
            Timer * pclLinkListNode_ )
```

Remove Remove a timer from the TimerList, cancelling its expiry.

Parameters

*pclLinkListNode_*	Pointer to the Timer to remove

Definition at line 51 of file timerlist.cpp.

19.39.3 Member Data Documentation

19.39.3.1 m_bTimerActive

```
bool Mark3::TimerList::m_bTimerActive [private]
```

Whether or not the timer is active.

Definition at line 78 of file timerlist.h.

19.39.3.2 m_clMutex

`Mutex Mark3::TimerList::m_clMutex [private]`

Guards against concurrent access to the timer list - Only needed when running threaded.

Definition at line 81 of file timerlist.h.

19.39.3.3 m_u32NextWakeup

`uint32_t Mark3::TimerList::m_u32NextWakeup [private]`

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

Definition at line 75 of file timerlist.h.

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

- /home/moslevin/projects/m3-repo/kernel/src/public/timerlist.h
- /home/moslevin/projects/m3-repo/kernel/src/timerlist.cpp

19.40 Mark3::TimerScheduler Class Reference

The TimerScheduler Class. This implements a "Static" class used to manage a global list of timers used throughout the system.

`#include <timerscheduler.h>`

Static Public Member Functions

- static void Init ()

  Init Initialize the timer scheduler. Must be called before any timer, or timer-derived functions are used.
- static void Add (Timer ∗pclListNode_)

  Add Add a timer to the timer scheduler. Adding a timer implicitly starts the timer as well.
- static void Remove (Timer ∗pclListNode_)

  Remove Remove a timer from the timer scheduler. May implicitly stop the timer if this is the only active timer scheduled.
- static void Process ()

  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.

Static Private Attributes

- static TimerList m_clTimerList

  TimerList object manipu32ated by the Timer Scheduler.

19.40.1 Detailed Description

The TimerScheduler Class. This implements a "Static" class used to manage a global list of timers used throughout the system.

Definition at line 38 of file timerscheduler.h.

19.40.2 Member Function Documentation

19.40.2.1 Add()

```
static void Mark3::TimerScheduler::Add (
             Timer * pclListNode_ )  [inline], [static]
```

Add Add a timer to the timer scheduler. Adding a timer implicitly starts the timer as well.

Parameters

*pclList←Node_*	Pointer to the timer list node to add

Definition at line 54 of file timerscheduler.h.

19.40.2.2 Init()

```
static void Mark3::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 46 of file timerscheduler.h.

19.40.2.3 Process()

```
static void Mark3::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 70 of file timerscheduler.h.

19.40.2.4 Remove()

```
static void Mark3::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

*pclList←↩ Node_*	Pointer to the timer list node to remove

Definition at line 62 of file timerscheduler.h.

19.40.3 Member Data Documentation

19.40.3.1 m_clTimerList

```
TimerList Mark3::TimerScheduler::m_clTimerList  [static], [private]
```

TimerList object manipu32ated by the Timer Scheduler.

Definition at line 74 of file timerscheduler.h.

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

- /home/moslevin/projects/m3-repo/kernel/src/public/timerscheduler.h
- /home/moslevin/projects/m3-repo/kernel/src/timer.cpp

19.41 Mark3::Token_t Struct Reference

Token descriptor struct format.

```
#include <memutil.h>
```

Public Attributes

- const char ∗ pcToken

  Pointer to the beginning of the token string.
- uint8_t u8Len

  Length of the token (in bytes)

19.41.1 Detailed Description

Token descriptor struct format.

Definition at line 32 of file memutil.h.

19.41.2 Member Data Documentation

19.41.2.1 pcToken

```
const char* Mark3::Token_t::pcToken
```

Pointer to the beginning of the token string.

Definition at line 33 of file memutil.h.

19.41.2.2 u8Len

```
uint8_t Mark3::Token_t::u8Len
```

Length of the token (in bytes)

Definition at line 34 of file memutil.h.

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

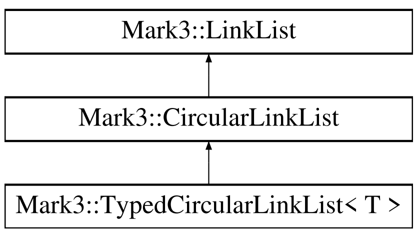
- /home/moslevin/projects/m3-repo/kernel/lib/memutil/public/memutil.h

19.42 Mark3::TypedCircularLinkList$<$ T $>$ Class Template Reference

The TypedCircularLinkList Class Circular-linked-list data type, inherited from the base LinkList type, and templated for use with linked-list-node derived data-types.

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

Inheritance diagram for Mark3::TypedCircularLinkList$<$ T $>$:



Public Member Functions

- void ∗ operator new (size_t sz, void ∗pv)
- TypedCircularLinkList ()
- T ∗ GetHead ()

  GetHead Get the head node in the linked list.
- void SetHead (T ∗pclNode_)

  SetHead Set the head node of a linked list.
- T ∗ GetTail ()

  GetTail Get the tail node of the linked list.
- void SetTail (T ∗pclNode_)

  SetTail Set the tail node of the linked list.
- void Add (T ∗pNode_)

  Add the linked list node to this linked list.
- void Remove (T ∗pNode_)

  Remove Add the linked list node to this linked list.
- void InsertNodeBefore (T ∗pNode_, T ∗pInsert_)

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

Additional Inherited Members

19.42.1 Detailed Description

template<typename T>
class Mark3::TypedCircularLinkList< T >

The TypedCircularLinkList Class Circular-linked-list data type, inherited from the base LinkList type, and templated for use with linked-list-node derived data-types.

Definition at line 347 of file ll.h.

19.42.2 Constructor & Destructor Documentation

19.42.2.1 TypedCircularLinkList()

```
template<typename T>
Mark3::TypedCircularLinkList< T >::TypedCircularLinkList ( )  [inline]
```

Definition at line 352 of file ll.h.

19.42.3 Member Function Documentation

19.42.3.1 Add()

```
template<typename T>
void Mark3::TypedCircularLinkList< T >::Add (
            T * pNode_ )  [inline]
```

Add the linked list node to this linked list.

Parameters

node←↩ _	Pointer to the node to add

Definition at line 394 of file ll.h.

19.42.3.2 GetHead()

```
template<typename T>
T* Mark3::TypedCircularLinkList< T >::GetHead ( ) [inline]
```

GetHead Get the head node in the linked list.

Returns

Pointer to the head node in the list

Definition at line 362 of file ll.h.

19.42.3.3 GetTail()

```
template<typename T>
T* Mark3::TypedCircularLinkList< T >::GetTail ( ) [inline]
```

GetTail Get the tail node of the linked list.

Returns

Pointer to the tail node in the list

Definition at line 378 of file ll.h.

19.42.3.4 InsertNodeBefore()

```
template<typename T>
void Mark3::TypedCircularLinkList< T >::InsertNodeBefore (
            T * pNode_,
            T * pInsert_ ) [inline]
```

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 418 of file ll.h.

19.42.3.5 operator new()

```
template<typename T>
void* Mark3::TypedCircularLinkList< T >::operator new (
            size_t sz,
            void * pv ) [inline]
```

Definition at line 350 of file ll.h.

19.42.3.6 Remove()

```
template<typename T>
void Mark3::TypedCircularLinkList< T >::Remove (
            T * pNode_ ) [inline]
```

Remove Add the linked list node to this linked list.

Parameters

*node_*	Pointer to the node to remove

Definition at line 405 of file ll.h.

19.42.3.7 SetHead()

```
template<typename T>
void Mark3::TypedCircularLinkList< T >::SetHead (
            T * pclNode_ ) [inline]
```

SetHead Set the head node of a linked list.

Parameters

*pcl←- Node_*	Pointer to node to set as the head of the linked list

Definition at line 370 of file ll.h.

19.42.3.8 SetTail()

```
template<typename T>
void Mark3::TypedCircularLinkList< T >::SetTail (
            T * pclNode_ )  [inline]
```

SetTail Set the tail node of the linked list.

Parameters

*pcl←- Node_*	Pointer to the node to set as the tail of the linked list

Definition at line 386 of file ll.h.

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

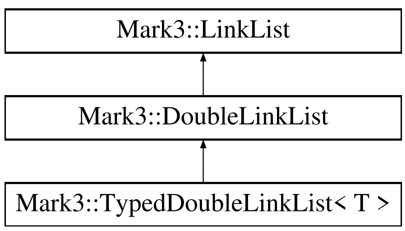
- /home/moslevin/projects/m3-repo/kernel/src/public/ll.h

19.43 Mark3::TypedDoubleLinkList< T > Class Template Reference

The TypedDoubleLinkList Class Doubly-linked-list data type, inherited from the base LinkList type, and templated for use with linked-list-node derived data-types.

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

Inheritance diagram for Mark3::TypedDoubleLinkList< T >:



Public Member Functions

- void ∗ operator new (size_t sz, void ∗pv)
- TypedDoubleLinkList ()
- T ∗ GetHead ()

    GetHead Get the head node in the linked list.
- void SetHead (T ∗pclNode_)

    SetHead Set the head node of a linked list.
- T ∗ GetTail ()

    GetTail Get the tail node of the linked list.
- void SetTail (T ∗pclNode_)

    SetTail Set the tail node of the linked list.
- void Add (T ∗pNode_)

    Add Add the linked list node to this linked list.
- void Remove (T ∗pNode_)

    Remove Add the linked list node to this linked list.

Additional Inherited Members

19.43.1 Detailed Description

template<typename T>
class Mark3::TypedDoubleLinkList< T >

The TypedDoubleLinkList Class Doubly-linked-list data type, inherited from the base LinkList type, and templated for use with linked-list-node derived data-types.

Definition at line 276 of file ll.h.

19.43.2 Constructor & Destructor Documentation

19.43.2.1 TypedDoubleLinkList()

```
template<typename T>
Mark3::TypedDoubleLinkList< T >::TypedDoubleLinkList ( )  [inline]
```

Definition at line 281 of file ll.h.

19.43.3 Member Function Documentation

19.43.3.1 Add()

```
template<typename T>
void Mark3::TypedDoubleLinkList< T >::Add (
            T * pNode_ )  [inline]
```

Add Add the linked list node to this linked list.

Parameters

node←→_	Pointer to the node to add

Definition at line 323 of file ll.h.

19.43.3.2 GetHead()

```
template<typename T>
T* Mark3::TypedDoubleLinkList< T >::GetHead ( ) [inline]
```

GetHead Get the head node in the linked list.

Returns

Pointer to the head node in the list

Definition at line 291 of file ll.h.

19.43.3.3 GetTail()

```
template<typename T>
T* Mark3::TypedDoubleLinkList< T >::GetTail ( ) [inline]
```

GetTail Get the tail node of the linked list.

Returns

Pointer to the tail node in the list

Definition at line 307 of file ll.h.

19.43.3.4 operator new()

```
template<typename T>
void* Mark3::TypedDoubleLinkList< T >::operator new (
            size_t sz,
            void * pv ) [inline]
```

Definition at line 279 of file ll.h.

19.43.3.5 Remove()

```
template<typename T>
void Mark3::TypedDoubleLinkList< T >::Remove (
            T * pNode_ ) [inline]
```

Remove Add the linked list node to this linked list.

Parameters

node←↩ _	Pointer to the node to remove

Definition at line 334 of file ll.h.

19.43.3.6 SetHead()

```
template<typename T>
void Mark3::TypedDoubleLinkList< T >::SetHead (
            T * pclNode_ )  [inline]
```

SetHead Set the head node of a linked list.

Parameters

*pcl←↩ Node_*	Pointer to node to set as the head of the linked list

Definition at line 299 of file ll.h.

19.43.3.7 SetTail()

```
template<typename T>
void Mark3::TypedDoubleLinkList< T >::SetTail (
            T * pclNode_ )  [inline]
```

SetTail Set the tail node of the linked list.

Parameters

*pcl←↩ Node_*	Pointer to the node to set as the tail of the linked list

Definition at line 315 of file ll.h.

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

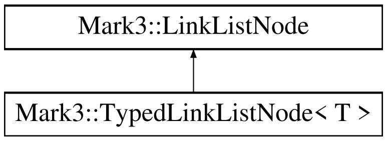
- /home/moslevin/projects/m3-repo/kernel/src/public/ll.h

19.44 Mark3::TypedLinkListNode< T > Class Template Reference

The TypedLinkListNode class The TypedLinkListNode class provides a linked-list node type for a specified object type. This can be used with typed link-list data structures to manage lists of objects without having to static-cast between the base type and the derived class.

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

Inheritance diagram for Mark3::TypedLinkListNode< T >:



Public Member Functions

- T ∗ GetNext ()
- T ∗ GetPrev ()

Additional Inherited Members

19.44.1 Detailed Description

template<typename T>
class Mark3::TypedLinkListNode< T >

The TypedLinkListNode class The TypedLinkListNode class provides a linked-list node type for a specified object type. This can be used with typed link-list data structures to manage lists of objects without having to static-cast between the base type and the derived class.

Definition at line 108 of file ll.h.

19.44.2 Member Function Documentation

19.44.2.1 GetNext()

```
template<typename T>
T* Mark3::TypedLinkListNode< T >::GetNext (
              void ) [inline]
```

Definition at line 111 of file ll.h.

19.44.2.2 GetPrev()

```
template<typename T>
T* Mark3::TypedLinkListNode< T >::GetPrev (
              void ) [inline]
```

Definition at line 112 of file ll.h.

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

- /home/moslevin/projects/m3-repo/kernel/src/public/ll.h

Chapter 20

File Documentation

20.1 /home/moslevin/projects/m3-repo/kernel/lib/mark3c/src/mark3c.cpp File Reference

Implementation of C-language wrappers for the Mark3 kernel.

```
#include "mark3c.h"
#include "mark3.h"
```

Functions

- void ∗ Alloc_Memory (uint16_t u16Size_)
- void Free_Memory (void ∗pvObject_)

  Free_Memory.
- Semaphore_t Alloc_Semaphore (void)

  Alloc_Semaphore.
- void Free_Semaphore (Semaphore_t handle)
- Mutex_t Alloc_Mutex (void)

  Alloc_Mutex.
- void Free_Mutex (Mutex_t handle)
- EventFlag_t Alloc_EventFlag (void)
- void Free_EventFlag (EventFlag_t handle)
- Message_t Alloc_Message (void)

  Alloc_Message.
- void Free_Message (Message_t handle)
- MessageQueue_t Alloc_MessageQueue (void)

  Alloc_MessageQueue.
- void Free_MessageQueue (MessageQueue_t handle)
- MessagePool_t Alloc_MessagePool (void)
- void Free_MessagePool (MessagePool_t handle)
- Notify_t Alloc_Notify (void)

  Alloc_Notify.
- void Free_Notify (Notify_t handle)
- Mailbox_t Alloc_Mailbox (void)

  Alloc_Mailbox.
- void Free_Mailbox (Mailbox_t handle)

- ConditionVariable_t Alloc_ConditionVariable (void)
- void Free_ConditionVariable (ConditionVariable_t handle)
- ReaderWriterLock_t Alloc_ReaderWriterLock (void)
- void Free_ReaderWriterLock (ReaderWriterLock_t handle)
- Thread_t Alloc_Thread (void)

    Alloc_Thread.
- void Free_Thread (Thread_t handle)
- Timer_t Alloc_Timer (void)

    Alloc_Timer.
- void Free_Timer (Timer_t handle)
- void Kernel_Init (void)

    Kernel_Init.
- void Kernel_Start (void)

    Kernel_Start.
- bool Kernel_IsStarted (void)

    Kernel_IsStarted.
- void Kernel_SetPanic (PanicFunc pfPanic_)

    Kernel_SetPanic.
- bool Kernel_IsPanic (void)

    Kernel_IsPanic.
- void Kernel_Panic (uint16_t u16Cause_)

    Kernel_Panic.
- uint32_t Kernel_GetTicks (void)

    Kernel_GetTicks.
- void Kernel_SetStackGuardThreshold (uint16_t u16Threshold_)
- uint16_t Kernel_GetStackGuardThreshold (void)
- void Kernel_SetDebugPrintFunction (kernel_debug_print_t pfPrintFunction_)

    Kernel_SetDebugPrintFunction.
- void Kernel_DebugPrint (const char ∗szString_)

    KernelDebug_DebugPrint.
- 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_, ThreadEntryFunc pfEntryPoint_, void ∗pvArg_)

    Thread_Init.
- void Thread_Start (Thread_t handle)

    Thread_Start.
- void Thread_Stop (Thread_t handle)

    Thread_Stop.
- void Thread_SetName (Thread_t handle, const char ∗szName_)
- const char ∗ Thread_GetName (Thread_t handle)
- 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_)
- uint16_t Thread_GetQuantum (Thread_t handle)
- 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_Yield (void)

  Thread_Yield.
- void Thread_CoopYield (void)

  Thread_CoopYield.
- 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_state_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_, timer_callback_t pfCallback_, void ∗pvData_)

  Timer_Start.
- void Timer_Stop (Timer_t handle)

  Timer_Stop.
- void Timer_Restart (Timer_t handle)

  Timer_Restart.
- 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 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.
- uint8_t Atomic_Set8 (uint8_t ∗pu8Source_, uint8_t u8Val_)

  Atomic_Set8.

- uint16_t Atomic_Set16 (uint16_t *pu16Source_, uint16_t u16Val_)

  Atomic_Set16.
- uint32_t Atomic_Set32 (uint32_t *pu32Source_, uint32_t u32Val_)

  Atomic_Set32.
- uint8_t Atomic_Add8 (uint8_t *pu8Source_, uint8_t u8Val_)

  Atomic_Add8.
- uint16_t Atomic_Add16 (uint16_t *pu16Source_, uint16_t u16Val_)

  Atomic_Add16.
- uint32_t Atomic_Add32 (uint32_t *pu32Source_, uint32_t u32Val_)

  Atomic_Add32.
- uint8_t Atomic_Sub8 (uint8_t *pu8Source_, uint8_t u8Val_)

  Atomic_Sub8.
- uint16_t Atomic_Sub16 (uint16_t *pu16Source_, uint16_t u16Val_)

  Atomic_Sub16.
- uint32_t Atomic_Sub32 (uint32_t *pu32Source_, uint32_t u32Val_)

  Atomic_Sub32.
- bool Atomic_TestAndSet (bool *pbLock)

  Atomic_TestAndSet.
- 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 MessageQueue_Init (MessageQueue_t handle)

  MessageQueue_Init.
- Message_t MessageQueue_Receive (MessageQueue_t handle)

  MessageQueue_Receive.
- void MessagePool_Init (MessagePool_t handle)

  MessagePool_Init.
- void MessagePool_Push (MessagePool_t handle, Message_t msg)

  MessagePool_Push.
- Message_t MessagePool_Pop (MessagePool_t handle)

  MessagePool_Pop.
- 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 (MessageQueue_t handle)

  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 ConditionVariable_Init (ConditionVariable_t handle)

  ConditionVariable_Init.
- void ConditionVariable_Wait (ConditionVariable_t handle, Mutex_t hMutex_)

  ConditionVariable_Wait.
- void ConditionVariable_Signal (ConditionVariable_t handle)

  ConditionVariable_Signal.
- void ConditionVariable_Broadcast (ConditionVariable_t handle)

  ConditionVariable_Broadcast.
- bool ConditionVariable_TimedWait (ConditionVariable_t handle, Mutex_t hMutex_, uint32_t u32WaitTime↩ MS_)

  ConditionVariable_TimedWait.
- void ReaderWriterLock_Init (ReaderWriterLock_t handle)

  ReaderWriterLock_Init.
- void ReaderWriterLock_AcquireReader (ReaderWriterLock_t handle)

  ReaderWriterLock_AcquireReader.
- void ReaderWriterLock_ReleaseReader (ReaderWriterLock_t handle)

  ReaderWriterLock_ReleaseReader.
- void ReaderWriterLock_AcquireWriter (ReaderWriterLock_t handle)

  ReaderWriterLock_AcquireWriter.
- void ReaderWriterLock_ReleaseWriter (ReaderWriterLock_t handle)

  ReaderWriterLock_ReleaseWriter.
- bool ReaderWriterLock_TimedAcquireWriter (ReaderWriterLock_t handle, uint32_t u32TimeoutMs_)

  ReaderWriterLock_TimedAcquireWriter.
- bool ReaderWriterLock_TimedAcquireReader (ReaderWriterLock_t handle, uint32_t u32TimeoutMs_)

  ReaderWriterLock_TimedAcquireReader.

20.1.1 Detailed Description

Implementation of C-language wrappers for the Mark3 kernel.

Definition in file mark3c.cpp.

20.1.2 Function Documentation

20.1.2.1 Alloc_ConditionVariable()

```
ConditionVariable_t Alloc_ConditionVariable (
            void )
```

Definition at line 134 of file mark3c.cpp.

20.1.2.2 Alloc_EventFlag()

```
EventFlag_t Alloc_EventFlag (
            void )
```

Definition at line 64 of file mark3c.cpp.

20.1.2.3 Alloc_Mailbox()

```
Mailbox_t Alloc_Mailbox (
            void )
```

Alloc_Mailbox.

See also

Mailbox* AutoAlloc::NewMailbox()

Returns

Handle to an allocated object, or nullptr if heap exhausted

Definition at line 123 of file mark3c.cpp.

20.1.2.4 Alloc_Memory()

```
void* Alloc_Memory (
            uint16_t u16Size_ )
```

Definition at line 29 of file mark3c.cpp.

20.1.2.5 Alloc_Message()

```
Message_t Alloc_Message (
            void )
```

Alloc_Message.

See also

AutoAlloc::NewMessage()

Returns

Handle to an allocated object, or nullptr if heap exhausted

Definition at line 76 of file mark3c.cpp.

20.1.2.6 Alloc_MessagePool()

```
MessagePool_t Alloc_MessagePool (
            void )
```

Definition at line 100 of file mark3c.cpp.

20.1.2.7 Alloc_MessageQueue()

```
MessageQueue_t Alloc_MessageQueue (
            void )
```

Alloc_MessageQueue.

See also

MesageQueue∗ AutoAlloc::NewMessageQueue()

Returns

Handle to an allocated object, or nullptr if heap exhausted

Definition at line 88 of file mark3c.cpp.

20.1.2.8 Alloc_Mutex()

```
Mutex_t Alloc_Mutex (
            void )
```

Alloc_Mutex.

See also

Mutex∗ AutoAlloc::NewMutex()

Returns

Handle to an allocated object, or nullptr if heap exhausted

Definition at line 52 of file mark3c.cpp.

20.1.2.9 Alloc_Notify()

```
Notify_t Alloc_Notify (
            void )
```

Alloc_Notify.

See also

Notify∗ AutoAlloc::NewNotify()

Returns

Handle to an allocated object, or nullptr if heap exhausted

Definition at line 112 of file mark3c.cpp.

20.1.2.10 Alloc_ReaderWriterLock()

```
ReaderWriterLock_t Alloc_ReaderWriterLock (
            void )
```

Definition at line 145 of file mark3c.cpp.

20.1.2.11 Alloc_Semaphore()

```
Semaphore_t Alloc_Semaphore (
            void )
```

Alloc_Semaphore.

See also

Semaphore∗ AutoAlloc::NewSemaphore()

Returns

Handle to an allocated object, or nullptr if heap exhausted

Definition at line 40 of file mark3c.cpp.

20.1.2.12 Alloc_Thread()

```
Thread_t Alloc_Thread (
            void )
```

Alloc_Thread.

See also

Thread∗ AutoAlloc::NewThread()

Returns

Handle to an allocated object, or nullptr if heap exhausted

Definition at line 156 of file mark3c.cpp.

20.1.2.13 Alloc_Timer()

```
Timer_t Alloc_Timer (
            void )
```

Alloc_Timer.

See also

Timer∗ AutoAlloc::NewTimer()

Returns

Handle to an allocated object, or nullptr if heap exhausted

Definition at line 167 of file mark3c.cpp.

20.1.2.14 Atomic_Add16()

```
uint16_t Atomic_Add16 (
            uint16_t * pu16Source_,
            uint16_t u16Val_ )
```

Atomic_Add16.

See also

uint16_t Atomic::Add(uint16_t *pu16Source_, uint16_t u16Val_)

Parameters

*pu16←→ Source_*	Pointer to a variable
*u16Val_*	Value to add to the variable

Returns

Previously-held value in pu16Source_

Definition at line 638 of file mark3c.cpp.

20.1.2.15 Atomic_Add32()

```
uint32_t Atomic_Add32 (
            uint32_t * pu32Source_,
            uint32_t u32Val_ )
```

Atomic_Add32.

See also

uint32_t Atomic::Add(uint32_t *pu32Source_, uint32_t u32Val_)

Parameters

*pu32←→ Source_*	Pointer to a variable
*u32Val_*	Value to add to the variable

Returns

Previously-held value in pu32Source_

Definition at line 644 of file mark3c.cpp.

20.1.2.16 Atomic_Add8()

```
uint8_t Atomic_Add8 (
            uint8_t * pu8Source_,
            uint8_t u8Val_ )
```

Atomic_Add8.

See also

uint8_t Atomic::Add(uint8_t *pu8Source_, uint8_t u8Val_)

Parameters

*pu8Source_*	Pointer to a variable
*u8Val_*	Value to add to the variable

Returns

Previously-held value in pu8Source_

Definition at line 632 of file mark3c.cpp.

20.1.2.17 Atomic_Set16()

```
uint16_t Atomic_Set16 (
            uint16_t * pu16Source_,
            uint16_t u16Val_ )
```

Atomic_Set16.

See also

uint16_t Atomic::Set(uint16_t *pu16Source_, uint16_t u16Val_)

Parameters

*pu16Source_*	Pointer to a variable to set the value of
*u16Val_*	New value to set in the variable

Returns

Previously-set value

Definition at line 620 of file mark3c.cpp.

20.1.2.18 Atomic_Set32()

```
uint32_t Atomic_Set32 (
            uint32_t * pu32Source_,
            uint32_t u32Val_ )
```

Atomic_Set32.

See also

uint32_t Atomic::Set(uint32_t *pu32Source_, uint32_t u32Val_)

Parameters

*pu32↩ Source_*	Pointer to a variable to set the value of
*u32Val_*	New value to set in the variable

Returns

Previously-set value

Definition at line 626 of file mark3c.cpp.

20.1.2.19 Atomic_Set8()

```
uint8_t Atomic_Set8 (
            uint8_t * pu8Source_,
            uint8_t u8Val_ )
```

Atomic_Set8.

See also

uint8_t Atomic::Set(uint8_t *pu8Source_, uint8_t u8Val_)

Parameters

*pu8↩ Source_*	Pointer to a variable to set the value of
*u8Val_*	New value to set in the variable

Returns

Previously-set value

Definition at line 614 of file mark3c.cpp.

20.1.2.20 Atomic_Sub16()

```
uint16_t Atomic_Sub16 (
            uint16_t * pu16Source_,
            uint16_t u16Val_ )
```

Atomic_Sub16.

See also

uint16_t Atomic::Sub(uint16_t *pu16Source_, uint16_t u16Val_)

Parameters

*pu16Source_*	Pointer to a variable
*u16Val_*	Value to subtract from the variable

Returns

Previously-held value in pu16Source_

Definition at line 656 of file mark3c.cpp.

20.1.2.21 Atomic_Sub32()

```
uint32_t Atomic_Sub32 (
            uint32_t * pu32Source_,
            uint32_t u32Val_ )
```

Atomic_Sub32.

See also

uint32_t Atomic::Sub(uint32_t *pu32Source_, uint32_t u32Val_)

Parameters

*pu32Source_*	Pointer to a variable
*u32Val_*	Value to subtract from the variable

Returns

Previously-held value in pu32Source_

Definition at line 662 of file mark3c.cpp.

20.1.2.22 Atomic_Sub8()

```
uint8_t Atomic_Sub8 (
            uint8_t * pu8Source_,
            uint8_t u8Val_ )
```

Atomic_Sub8.

See also

uint8_t Atomic::Sub(uint8_t *pu8Source_, uint8_t u8Val_)

Parameters

*pu8←Source_*	Pointer to a variable
*u8Val_*	Value to subtract from the variable

Returns

Previously-held value in pu8Source_

Definition at line 650 of file mark3c.cpp.

20.1.2.23 Atomic_TestAndSet()

```
bool Atomic_TestAndSet (
            bool * pbLock )
```

Atomic_TestAndSet.

See also

bool Atomic::TestAndSet(bool *pbLock)

Parameters

pbLock	Pointer to a value to test against. This will always be set to "true" at the end of a call to TestAndSet.

Returns

true - Lock value was "true" on entry, false - Lock was set

Definition at line 668 of file mark3c.cpp.

20.1.2.24 ConditionVariable_Broadcast()

```
void ConditionVariable_Broadcast (
            ConditionVariable_t handle )
```

ConditionVariable_Broadcast.

See also

void ConditionVariable::Broadcast()

Parameters

handle	Handle of the condition variable object

Definition at line 874 of file mark3c.cpp.

20.1.2.25 ConditionVariable_Init()

```
void ConditionVariable_Init (
            ConditionVariable_t handle )
```

ConditionVariable_Init.

See also

void ConditionVariable::Init()

Parameters

handle	Handle of the condition variable object

Definition at line 855 of file mark3c.cpp.

20.1.2.26 ConditionVariable_Signal()

```
void ConditionVariable_Signal (
            ConditionVariable_t handle )
```

ConditionVariable_Signal.

See also

void ConditionVariable::Signal()

Parameters

handle	Handle of the condition variable object

Definition at line 868 of file mark3c.cpp.

20.1.2.27 ConditionVariable_TimedWait()

```
bool ConditionVariable_TimedWait (
            ConditionVariable_t handle,
            Mutex_t hMutex_,
            uint32_t u32WaitTimeMS_ )
```

ConditionVariable_TimedWait.

See also

bool ConditionVariable::Wait(Mutex∗ pclMutex_, uint32_t u32WaitTimeMS_)

Parameters

handle	Handle of the condition variable object
*hMutex_*	Handle of the mutex to lock on acquisition of the condition variable
*u32WaitTimeMS_*	Maximum time to wait for object

Returns

true on success, false on timeout

Definition at line 880 of file mark3c.cpp.

20.1.2.28 ConditionVariable_Wait()

```
void ConditionVariable_Wait (
            ConditionVariable_t handle,
            Mutex_t hMutex_ )
```

ConditionVariable_Wait.

See also

void ConditionVariable::Wait(Mutex∗ pclMutex_)

Parameters

handle	Handle of the condition variable object
h← Mutex← _	Handle of the mutex to lock on acquisition of the condition variable

Definition at line 861 of file mark3c.cpp.

20.1.2.29 Free_ConditionVariable()

```
void Free_ConditionVariable (
            ConditionVariable_t handle )
```

Definition at line 139 of file mark3c.cpp.

20.1.2.30 Free_EventFlag()

```
void Free_EventFlag (
            EventFlag_t handle )
```

Definition at line 70 of file mark3c.cpp.

20.1.2.31 Free_Mailbox()

```
void Free_Mailbox (
            Mailbox_t handle )
```

Definition at line 128 of file mark3c.cpp.

20.1.2.32 Free_Memory()

```
void Free_Memory (
            void * pvObject_ )
```

Free_Memory.

Parameters

*pv← Object_*	Pointer to previously allocated block of memory

Definition at line 34 of file mark3c.cpp.

20.1.2.33 Free_Message()

```
void Free_Message (
            Message_t handle )
```

Definition at line 82 of file mark3c.cpp.

20.1.2.34 Free_MessagePool()

```
void Free_MessagePool (
            MessagePool_t handle )
```

Definition at line 106 of file mark3c.cpp.

20.1.2.35 Free_MessageQueue()

```
void Free_MessageQueue (
            MessageQueue_t handle )
```

Definition at line 94 of file mark3c.cpp.

20.1.2.36 Free_Mutex()

```
void Free_Mutex (
            Mutex_t handle )
```

Definition at line 58 of file mark3c.cpp.

20.1.2.37 Free_Notify()

```
void Free_Notify (
            Notify_t handle )
```

Definition at line 118 of file mark3c.cpp.

20.1.2.38 Free_ReaderWriterLock()

```
void Free_ReaderWriterLock (
            ReaderWriterLock_t handle )
```

Definition at line 150 of file mark3c.cpp.

20.1.2.39 Free_Semaphore()

```
void Free_Semaphore (
            Semaphore_t handle )
```

Definition at line 46 of file mark3c.cpp.

20.1.2.40 Free_Thread()

```
void Free_Thread (
            Thread_t handle )
```

Definition at line 161 of file mark3c.cpp.

20.1.2.41 Free_Timer()

```
void Free_Timer (
            Timer_t handle )
```

Definition at line 172 of file mark3c.cpp.

20.1.2.42 Kernel_DebugPrint()

```
void Kernel_DebugPrint (
            const char * szString_ )
```

KernelDebug_DebugPrint.

See also

void DebugPrint(const char∗ szString_)

Parameters

sz↩ *String_*	String to print to debug interface

Definition at line 282 of file mark3c.cpp.

20.1.2.43 Kernel_GetStackGuardThreshold()

```
uint16_t Kernel_GetStackGuardThreshold (
            void  )
```

Definition at line 270 of file mark3c.cpp.

20.1.2.44 Kernel_GetTicks()

```
uint32_t Kernel_GetTicks (
            void  )
```

Kernel_GetTicks.

See also

Kernel::GetTicks()

Returns

Number of kernel ticks that have elapsed since boot

Definition at line 216 of file mark3c.cpp.

20.1.2.45 Kernel_Init()

```
void Kernel_Init (
            void  )
```

Kernel_Init.

See also

void Kernel::Init()

Definition at line 180 of file mark3c.cpp.

20.1.2.46 Kernel_IsPanic()

```
bool Kernel_IsPanic (
            void  )
```

Kernel_IsPanic.

See also

bool Kernel::IsPanic()

Returns

Whether or not the kernel is in a panic state

Definition at line 204 of file mark3c.cpp.

20.1.2.47 Kernel_IsStarted()

```
bool Kernel_IsStarted (
            void  )
```

Kernel_IsStarted.

See also

bool Kernel::IsStarted()

Returns

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

Definition at line 192 of file mark3c.cpp.

20.1.2.48 Kernel_Panic()

```
void Kernel_Panic (
            uint16_t u16Cause_ )
```

Kernel_Panic.

See also

void Kernel::Panic(uint16_t u16Cause_)

Parameters

*u16↩ Cause_*	Reason for the kernel panic

Definition at line 210 of file mark3c.cpp.

20.1.2.49 Kernel_SetDebugPrintFunction()

```
void Kernel_SetDebugPrintFunction (
            kernel_debug_print_t pfPrintFunction_ )
```

Kernel_SetDebugPrintFunction.

See also

void Kernel::SetDebugPrintFunction()

Parameters

*pfPrint↩ Function_*	Function to use to print debug information from the kernel

Definition at line 276 of file mark3c.cpp.

20.1.2.50 Kernel_SetPanic()

```
void Kernel_SetPanic (
            panic_func_t pfPanic_ )
```

Kernel_SetPanic.

See also

void Kernel::SetPanic(PanicFunc_t pfPanic_)

Parameters

pf↩ Panic↩ _	Panic function pointer

Definition at line 198 of file mark3c.cpp.

20.1.2.51 Kernel_SetStackGuardThreshold()

```
void Kernel_SetStackGuardThreshold (
            uint16_t u16Threshold_ )
```

Definition at line 264 of file mark3c.cpp.

20.1.2.52 Kernel_Start()

```
void Kernel_Start (
            void )
```

Kernel_Start.

See also

void Kernel::Start()

Definition at line 186 of file mark3c.cpp.

20.1.2.53 Mailbox_GetFreeSlots()

```
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

Definition at line 832 of file mark3c.cpp.

20.1.2.54 Mailbox_Init()

```
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

Definition at line 769 of file mark3c.cpp.

20.1.2.55 Mailbox_IsEmpty()

```
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

Definition at line 846 of file mark3c.cpp.

20.1.2.56 Mailbox_IsFull()

```
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

Definition at line 839 of file mark3c.cpp.

20.1.2.57 Mailbox_Receive()

```
void Mailbox_Receive (
            Mailbox_t handle,
            void * pvData_ )
```

Mailbox_Receive.

See also

void Mailbox::Receive(void *pvData_)

Parameters

handle	Handle of the mailbox object
*pv↩ Data_*	Pointer to a buffer that will have the envelope's contents copied into upon delivery.

Definition at line 804 of file mark3c.cpp.

20.1.2.58 Mailbox_ReceiveTail()

```
void Mailbox_ReceiveTail (
            Mailbox_t handle,
            void * pvData_ )
```

Mailbox_ReceiveTail.

See also

void Mailbox::ReceiveTail(void *pvData_)

Parameters

handle	Handle of the mailbox object
*pv←→ Data_*	Pointer to a buffer that will have the envelope's contents copied into upon delivery.

Definition at line 811 of file mark3c.cpp.

20.1.2.59 Mailbox_Send()

```
bool Mailbox_Send (
            Mailbox_t handle,
            void * pvData_ )
```

Mailbox_Send.

See also

bool Mailbox::Send(void *pvData_)

Parameters

handle	Handle of the mailbox object
*pv←→ Data_*	Pointer to the data object to send to the mailbox.

Returns

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

Definition at line 776 of file mark3c.cpp.

20.1.2.60 Mailbox_SendTail()

```
bool Mailbox_SendTail (
            Mailbox_t handle,
            void * pvData_ )
```

Mailbox_SendTail.

See also

bool Mailbox::SendTail(void *pvData_)

Parameters

handle	Handle of the mailbox object
*pv← Data_*	Pointer to the data object to send to the mailbox.

Returns

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

Definition at line 783 of file mark3c.cpp.

20.1.2.61 Mailbox_TimedReceive()

```
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.

Definition at line 818 of file mark3c.cpp.

20.1.2.62 Mailbox_TimedReceiveTail()

```
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.
*u32TimeoutMS_*	Maximum time to wait for delivery.

Returns

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

Definition at line 825 of file mark3c.cpp.

20.1.2.63 Mailbox_TimedSend()

```
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.
*u32TimeoutMS_*	Maximum time to wait for a free transmit slot

Returns

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

Definition at line 790 of file mark3c.cpp.

20.1.2.64 Mailbox_TimedSendTail()

```
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.
*u32TimeoutMS_*	Maximum time to wait for a free transmit slot

Returns

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

Definition at line 797 of file mark3c.cpp.

20.1.2.65 Message_GetCode()

```
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

Definition at line 704 of file mark3c.cpp.

20.1.2.66 Message_GetData()

```
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

Definition at line 690 of file mark3c.cpp.

20.1.2.67 Message_Init()

```
void Message_Init (
            Message_t handle )
```

Message_Init.

See also

void Message::Init()

Parameters

handle	Handle of the message object

Definition at line 676 of file mark3c.cpp.

20.1.2.68 Message_SetCode()

```
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

Definition at line 697 of file mark3c.cpp.

20.1.2.69 Message_SetData()

```
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

Definition at line 683 of file mark3c.cpp.

20.1.2.70 MessagePool_Init()

```
void MessagePool_Init (
            MessagePool_t handle )
```

MessagePool_Init.

See also

void MessagePool::Init()

Parameters

handle	Handle of the message pool object

Definition at line 725 of file mark3c.cpp.

20.1.2.71 MessagePool_Pop()

```
Message_t MessagePool_Pop (
            MessagePool_t handle )
```

MessagePool_Pop.

See also

Message∗ MessagePool::Pop()

Parameters

handle	Handle of the message pool object

Returns

Handle to a Message object, or nullptr on allocation error

Definition at line 739 of file mark3c.cpp.

20.1.2.72 MessagePool_Push()

```
void MessagePool_Push (
            MessagePool_t handle,
            Message_t msg )
```

MessagePool_Push.

See also

void MessagePool::Push(Message∗ pclMessage_)

Parameters

handle	Handle of the message pool object
msg	Message object to return back to the pool

Definition at line 732 of file mark3c.cpp.

20.1.2.73 MessageQueue_GetCount()

```
uint16_t MessageQueue_GetCount (
            MessageQueue_t handle )
```

MessageQueue_GetCount.

See also

uint16_t MessageQueue::GetCount()

Returns

Count of pending messages in the queue.

Definition at line 760 of file mark3c.cpp.

20.1.2.74 MessageQueue_Init()

```
void MessageQueue_Init (
            MessageQueue_t handle )
```

MessageQueue_Init.

See also

void MessageQueue::Init()

Parameters

handle	Handle to the message queue to initialize

Definition at line 711 of file mark3c.cpp.

20.1.2.75 MessageQueue_Receive()

```
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

Definition at line 718 of file mark3c.cpp.

20.1.2.76 MessageQueue_Send()

```
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

Definition at line 753 of file mark3c.cpp.

20.1.2.77 MessageQueue_TimedReceive()

```
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 nullptr on timeout.

Definition at line 746 of file mark3c.cpp.

20.1.2.78 Mutex_Claim()

```
void Mutex_Claim (
            Mutex_t handle )
```

Mutex_Claim.

See also

void Mutex::Claim()

Parameters

handle	Handle of the mutex

Definition at line 515 of file mark3c.cpp.

20.1.2.79 Mutex_Init()

```
void Mutex_Init (
            Mutex_t handle )
```

Mutex_Init.

See also

void Mutex::Init()

Parameters

handle	Handle of the mutex

Definition at line 508 of file mark3c.cpp.

20.1.2.80 Mutex_Release()

```
void Mutex_Release (
            Mutex_t handle )
```

Mutex_Release.

See also

void Mutex::Release()

Parameters

handle	Handle of the mutex

Definition at line 522 of file mark3c.cpp.

20.1.2.81 Mutex_TimedClaim()

```
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

Definition at line 529 of file mark3c.cpp.

20.1.2.82 Notify_Init()

```
void Notify_Init (
            Notify_t handle )
```

Notify_Init.

See also

void Notify::Init()

Parameters

handle	Handle of the notification object

Definition at line 584 of file mark3c.cpp.

20.1.2.83 Notify_Signal()

```
void Notify_Signal (
            Notify_t handle )
```

Notify_Signal.

See also

void Notify::Signal()

Parameters

handle	Handle of the notification object

Definition at line 591 of file mark3c.cpp.

20.1.2.84 Notify_TimedWait()

```
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

Definition at line 605 of file mark3c.cpp.

20.1.2.85 Notify_Wait()

```
void Notify_Wait (
            Notify_t handle,
            bool * pbFlag_ )
```

Notify_Wait.

See also

void Notify::Wait(bool *pbFlag_)

Parameters

handle	Handle of the notification object
*pb↩ Flag_*	Flag to set to true on notification

Definition at line 598 of file mark3c.cpp.

20.1.2.86 ReaderWriterLock_AcquireReader()

```
void ReaderWriterLock_AcquireReader (
            ReaderWriterLock_t handle )
```

ReaderWriterLock_AcquireReader.

See also

void ReaderWriterLock::AcquireReader()

Parameters

handle	Handle of the reader-writer object

Definition at line 897 of file mark3c.cpp.

20.1.2.87 ReaderWriterLock_AcquireWriter()

```
void ReaderWriterLock_AcquireWriter (
            ReaderWriterLock_t handle )
```

ReaderWriterLock_AcquireWriter.

See also

void ReaderWriterLock::AcquireWriter()

Parameters

handle	Handle of the reader-writer object

Definition at line 911 of file mark3c.cpp.

20.1.2.88 ReaderWriterLock_Init()

```
void ReaderWriterLock_Init (
            ReaderWriterLock_t handle )
```

ReaderWriterLock_Init.

See also

void ReaderWriterLock::Init()

Parameters

handle	Handle of the reader-writer object

Definition at line 890 of file mark3c.cpp.

20.1.2.89 ReaderWriterLock_ReleaseReader()

```
void ReaderWriterLock_ReleaseReader (
            ReaderWriterLock_t handle )
```

ReaderWriterLock_ReleaseReader.

See also

void ReaderWriterLock::ReleaseReader()

Parameters

handle	Handle of the reader-writer object

Definition at line 904 of file mark3c.cpp.

20.1.2.90 ReaderWriterLock_ReleaseWriter()

```
void ReaderWriterLock_ReleaseWriter (
            ReaderWriterLock_t handle )
```

ReaderWriterLock_ReleaseWriter.

See also

void ReaderWriterLock::ReleaseWriter()

Parameters

handle	Handle of the reader-writer object

Definition at line 918 of file mark3c.cpp.

20.1.2.91 ReaderWriterLock_TimedAcquireReader()

```
bool ReaderWriterLock_TimedAcquireReader (
            ReaderWriterLock_t handle,
            uint32_t u32TimeoutMs_ )
```

ReaderWriterLock_TimedAcquireReader.

See also

bool ReaderWriterLock::AcquireReader(uint32_t u32TimeoutMs_)

Parameters

handle	Handle of the reader-writer object
*u32Timeout↩ Ms_*	Maximum time to wait for the reader lock before bailing

Returns

true on success, false on timeout

Definition at line 932 of file mark3c.cpp.

20.1.2.92 ReaderWriterLock_TimedAcquireWriter()

```
bool ReaderWriterLock_TimedAcquireWriter (
            ReaderWriterLock_t handle,
            uint32_t u32TimeoutMs_ )
```

ReaderWriterLock_TimedAcquireWriter.

See also

bool ReaderWriterLock::AcquireWriter(uint32_t u32TimeoutMs_)

Parameters

handle	Handle of the reader-writer object
*u32Timeout↩ Ms_*	Maximum time to wait for the writer lock before bailing

Returns

true on success, false on timeout

Definition at line 925 of file mark3c.cpp.

20.1.2.93 Scheduler_Enable()

```
void Scheduler_Enable (
            bool bEnable_ )
```

Scheduler_Enable.

See also

void Scheduler::SetScheduler(bool bEnable_)

Parameters

b← Enable← _	true to enable, false to disable the scheduler

Definition at line 289 of file mark3c.cpp.

20.1.2.94 Scheduler_GetCurrentThread()

```
Thread_t Scheduler_GetCurrentThread (
            void )
```

Scheduler_GetCurrentThread.

See also

Thread∗ Scheduler::GetCurrentThread()

Returns

Handle of the currently-running thread

Definition at line 301 of file mark3c.cpp.

20.1.2.95 Scheduler_IsEnabled()

```
bool Scheduler_IsEnabled (
            void )
```

Scheduler_IsEnabled.

See also

bool Scheduler::IsEnabled()

Returns

true - scheduler enabled, false - disabled

Definition at line 295 of file mark3c.cpp.

20.1.2.96 Semaphore_Init()

```
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
*u16Max←↩ Val_*	Maximum value that can be held for a semaphore

Definition at line 478 of file mark3c.cpp.

20.1.2.97 Semaphore_Pend()

```
void Semaphore_Pend (
            Semaphore_t handle )
```

Semaphore_Pend.

See also

void Semaphore::Pend()

Parameters

handle	Handle of the semaphore

Definition at line 492 of file mark3c.cpp.

20.1.2.98 Semaphore_Post()

```
void Semaphore_Post (
            Semaphore_t handle )
```

Semaphore_Post.

See also

void Semaphore::Post()

Parameters

handle	Handle of the semaphore

Definition at line 485 of file mark3c.cpp.

20.1.2.99 Semaphore_TimedPend()

```
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
*u32WaitTimeMS_*	Time in ms to wait

Returns

true if semaphore was acquired, false on timeout

Definition at line 499 of file mark3c.cpp.

20.1.2.100 Thread_CoopYield()

```
void Thread_CoopYield (
            void )
```

Thread_CoopYield.

See also

void Thread::CoopYield()

Definition at line 415 of file mark3c.cpp.

20.1.2.101 Thread_Exit()

```
void Thread_Exit (
            Thread_t handle )
```

Thread_Exit.

See also

void Thread::Exit()

Parameters

handle	Handle of the thread

Definition at line 381 of file mark3c.cpp.

20.1.2.102 Thread_GetCurPriority()

```
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

Definition at line 354 of file mark3c.cpp.

20.1.2.103 Thread_GetID()

```
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

Definition at line 426 of file mark3c.cpp.

20.1.2.104 Thread_GetName()

```
const char* Thread_GetName (
             Thread_t handle )
```

Definition at line 341 of file mark3c.cpp.

20.1.2.105 Thread_GetPriority()

```
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

Definition at line 348 of file mark3c.cpp.

20.1.2.106 Thread_GetQuantum()

```
uint16_t Thread_GetQuantum (
             Thread_t handle )
```

Definition at line 367 of file mark3c.cpp.

20.1.2.107 Thread_GetStackSlack()

```
uint16_t Thread_GetStackSlack (
            Thread_t handle )
```

Definition at line 432 of file mark3c.cpp.

20.1.2.108 Thread_GetState()

```
thread_state_t Thread_GetState (
            Thread_t handle )
```

Thread_GetState.

See also

ThreadState Thread::GetState()

Parameters

handle	Handle of the thread

Returns

The thread's current execution state

Definition at line 438 of file mark3c.cpp.

20.1.2.109 Thread_Init()

```
void Thread_Init (
            Thread_t handle,
            K_WORD * pwStack_,
            uint16_t u16StackSize_,
            PORT_PRIO_TYPE uXPriority_,
            thread_entry_func_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
u16Stack← *Size_*	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.

Definition at line 310 of file mark3c.cpp.

20.1.2.110 Thread_SetID()

```
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
u8I← *D_*	ID To assign to the thread

Definition at line 420 of file mark3c.cpp.

20.1.2.111 Thread_SetName()

```
void Thread_SetName (
            Thread_t handle,
            const char * szName_ )
```

Definition at line 335 of file mark3c.cpp.

20.1.2.112 Thread_SetPriority()

```
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
*uX↩ Priority_*	New priority level

Definition at line 374 of file mark3c.cpp.

20.1.2.113 Thread_SetQuantum()

```
void Thread_SetQuantum (
            Thread_t handle,
            uint16_t u16Quantum_ )
```

Definition at line 361 of file mark3c.cpp.

20.1.2.114 Thread_Sleep()

```
void Thread_Sleep (
            uint32_t u32TimeMs_ )
```

Thread_Sleep.

See also

void Thread::Sleep(uint32_t u32TimeMs_)

Parameters

*u32Time↩ Ms_*	Time in ms to block the thread for

Definition at line 388 of file mark3c.cpp.

20.1.2.115 Thread_Start()

```
void Thread_Start (
            Thread_t handle )
```

Thread_Start.

See also

void Thread::Start()

Parameters

handle	Handle of the thread to start

Definition at line 322 of file mark3c.cpp.

20.1.2.116 Thread_Stop()

```
void Thread_Stop (
            Thread_t handle )
```

Thread_Stop.

See also

void Thread::Stop()

Parameters

handle	Handle of the thread to stop

Definition at line 329 of file mark3c.cpp.

20.1.2.117 Thread_Yield()

```
void Thread_Yield (
            void )
```

Thread_Yield.

See also

void Thread::Yield()

Definition at line 410 of file mark3c.cpp.

20.1.2.118 Timer_Init()

```
void Timer_Init (
            Timer_t handle )
```

Timer_Init.

See also

void Timer::Init()

Parameters

handle	Handle of the timer

Definition at line 448 of file mark3c.cpp.

20.1.2.119 Timer_Restart()

```
void Timer_Restart (
            Timer_t handle )
```

Timer_Restart.

See also

void Timer::Start()

Parameters

handle	Handle of the timer to restart.

Definition at line 469 of file mark3c.cpp.

20.1.2.120 Timer_Start()

```
void Timer_Start (
            Timer_t handle,
```

```
bool bRepeat_,
uint32_t u32IntervalMs_,
timer_callback_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
*u32Interval↩ Ms_*	Time in ms to expiry
*pfCallback_*	Callback to run on timer expiry
*pvData_*	Data to pass to the callback on expiry

Definition at line 455 of file mark3c.cpp.

20.1.2.121 Timer_Stop()

```
void Timer_Stop (
        Timer_t handle )
```

Timer_Stop.

See also

void Timer::Stop()

Parameters

handle	Handle of the timer

Definition at line 462 of file mark3c.cpp.

20.2 mark3c.cpp

```
/*==============================================================================
     _____     _____     _____     _____
 ___|    _|__ __|_    |__ __|__   |__ __| __  |__  ______
|    \  /  | ||    \    ||    |     ||  |/ /    ||___   |
|     \/   | ||     \   ||     \    ||     \    ||___   |
|__/\__/|__|_||__|\__\ __||__|\__\ __||__|\__\ __||______|
```

```
    |_____|     |_____|     |_____|     |_____|

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

Copyright (c) 2012 - 2019 m0slevin, all rights reserved.
See license.txt for more information
===========================================================================*/
#include "mark3c.h"
#include "mark3.h"

using namespace Mark3;

//---------------------------------------------------------------------------
// Kernel Memory managment APIs
//---------------------------------------------------------------------------
void* Alloc_Memory(uint16_t u16Size_)
{
    return AutoAlloc::NewRawData(u16Size_);
}
//---------------------------------------------------------------------------
void Free_Memory(void* pvObject_)
{
    AutoAlloc::DestroyRawData(pvObject_);
}

//---------------------------------------------------------------------------
Semaphore_t Alloc_Semaphore(void)
{
    return (Semaphore_t)AutoAlloc::NewObject<Semaphore, AutoAllocType::Semaphore>();
}

//---------------------------------------------------------------------------
void Free_Semaphore(Semaphore_t handle)
{
    AutoAlloc::DestroyObject<Semaphore, AutoAllocType::Semaphore>((Semaphore*)handle);
}

//---------------------------------------------------------------------------
Mutex_t Alloc_Mutex(void)
{
    return (Mutex_t)AutoAlloc::NewObject<Mutex, AutoAllocType::Mutex>();
}

//---------------------------------------------------------------------------
void Free_Mutex(Mutex_t handle)
{
    AutoAlloc::DestroyObject<Mutex, AutoAllocType::Mutex>((Mutex*)handle);
}

//---------------------------------------------------------------------------
EventFlag_t Alloc_EventFlag(void)
{
    return (EventFlag_t)AutoAlloc::NewObject<EventFlag, AutoAllocType::EventFlag>();
}

//---------------------------------------------------------------------------
void Free_EventFlag(EventFlag_t handle)
{
    AutoAlloc::DestroyObject<EventFlag, AutoAllocType::EventFlag>((EventFlag*)handle);
}

//---------------------------------------------------------------------------
Message_t Alloc_Message(void)
{
    return (Message_t)AutoAlloc::NewObject<Message, AutoAllocType::Message>();
}

//---------------------------------------------------------------------------
void Free_Message(Message_t handle)
{
    AutoAlloc::DestroyObject<Message, AutoAllocType::Message>((Message*)handle);
}

//---------------------------------------------------------------------------
MessageQueue_t Alloc_MessageQueue(void)
{
    return (MessageQueue_t)AutoAlloc::NewObject<MessageQueue, AutoAllocType::MessageQueue>();
}

//---------------------------------------------------------------------------
void Free_MessageQueue(MessageQueue_t handle)
{
    AutoAlloc::DestroyObject<MessageQueue, AutoAllocType::MessageQueue>((
MessageQueue*)handle);
}

//---------------------------------------------------------------------------
```

```
MessagePool_t Alloc_MessagePool(void)
{
    return (MessagePool_t)AutoAlloc::NewObject<MessagePool, AutoAllocType::MessagePool>();
}
//---------------------------------------------------------------------------
void Free_MessagePool(MessagePool_t handle)
{
    AutoAlloc::DestroyObject<MessagePool, AutoAllocType::MessagePool>((MessagePool*)handle);
}
//---------------------------------------------------------------------------
Notify_t Alloc_Notify(void)
{
    return (Notify_t)AutoAlloc::NewObject<Notify, AutoAllocType::Notify>();
}
//---------------------------------------------------------------------------
void Free_Notify(Notify_t handle)
{
    AutoAlloc::DestroyObject<Notify, AutoAllocType::Notify>((Notify*)handle);
}
//---------------------------------------------------------------------------
Mailbox_t Alloc_Mailbox(void)
{
    return (Mailbox_t)AutoAlloc::NewObject<Mailbox, AutoAllocType::MailBox>();
}
//---------------------------------------------------------------------------
void Free_Mailbox(Mailbox_t handle)
{
    AutoAlloc::DestroyObject<Mailbox, AutoAllocType::MailBox>((Mailbox*)handle);
}
//---------------------------------------------------------------------------
ConditionVariable_t Alloc_ConditionVariable(void)
{
    return (ConditionVariable_t)AutoAlloc::NewObject<ConditionVariable, AutoAllocType::ConditionVariable>();
}
//---------------------------------------------------------------------------
void Free_ConditionVariable(ConditionVariable_t handle)
{
    AutoAlloc::DestroyObject<ConditionVariable, AutoAllocType::ConditionVariable>((ConditionVariable*)handle);
}
//---------------------------------------------------------------------------
ReaderWriterLock_t Alloc_ReaderWriterLock(void)
{
    return (ReaderWriterLock_t)AutoAlloc::NewObject<ReaderWriterLock, AutoAllocType::ReaderWriterLock>();
}
//---------------------------------------------------------------------------
void Free_ReaderWriterLock(ReaderWriterLock_t handle)
{
    AutoAlloc::DestroyObject<ReaderWriterLock, AutoAllocType::ReaderWriterLock>((ReaderWriterLock*)handle);
}
//---------------------------------------------------------------------------
Thread_t Alloc_Thread(void)
{
    return (Thread_t)AutoAlloc::NewObject<Thread, AutoAllocType::Thread>();
}
//---------------------------------------------------------------------------
void Free_Thread(Thread_t handle)
{
    AutoAlloc::DestroyObject<Thread, AutoAllocType::Thread>((Thread*)handle);
}
//---------------------------------------------------------------------------
Timer_t Alloc_Timer(void)
{
    return (Thread_t)AutoAlloc::NewObject<Timer, AutoAllocType::Timer>();
}
//---------------------------------------------------------------------------
void Free_Timer(Timer_t handle)
{
    AutoAlloc::DestroyObject<Timer, AutoAllocType::Timer>((Timer*)handle);
}
//---------------------------------------------------------------------------
// Kernel APIs
//---------------------------------------------------------------------------
void Kernel_Init(void)
{
```

```
    Kernel::Init();
}

//---------------------------------------------------------------------------
void Kernel_Start(void)
{
    Kernel::Start();
}

//---------------------------------------------------------------------------
bool Kernel_IsStarted(void)
{
    return Kernel::IsStarted();
}

//---------------------------------------------------------------------------
void Kernel_SetPanic(PanicFunc pfPanic_)
{
    Kernel::SetPanic(pfPanic_);
}

//---------------------------------------------------------------------------
bool Kernel_IsPanic(void)
{
    return Kernel::IsPanic();
}

//---------------------------------------------------------------------------
void Kernel_Panic(uint16_t u16Cause_)
{
    Kernel::Panic(u16Cause_);
}

//---------------------------------------------------------------------------
uint32_t Kernel_GetTicks(void)
{
    return Kernel::GetTicks();
}

#if KERNEL_THREAD_CREATE_CALLOUT
//---------------------------------------------------------------------------
void Kernel_SetThreadCreateCallout(thread_create_callout_t pfCreate_)
{
    Kernel::SetThreadCreateCallout((
ThreadCreateCallout)pfCreate_);
}
#endif // #if KERNEL_THREAD_CREATE_CALLOUT
//---------------------------------------------------------------------------
#if KERNEL_THREAD_EXIT_CALLOUT
void Kernel_SetThreadExitCallout(thread_exit_callout_t pfExit_)
{
    Kernel::SetThreadExitCallout((ThreadExitCallout)pfExit_);
}
#endif //#if KERNEL_THREAD_EXIT_CALLOUT
#if KERNEL_CONTEXT_SWITCH_CALLOUT
//---------------------------------------------------------------------------
void Kernel_SetThreadContextSwitchCallout(thread_context_callout_t pfContext_)
{
    Kernel::SetThreadContextSwitchCallout((
ThreadContextCallout)pfContext_);
}
#endif // #if KERNEL_CONTEXT_SWITCH_CALLOUT
#if KERNEL_THREAD_CREATE_CALLOUT
//---------------------------------------------------------------------------
thread_create_callout_t Kernel_GetThreadCreateCallout(void)
{
    return (thread_create_callout_t)Kernel::GetThreadCreateCallout();
}
#endif // #if KERNEL_THREAD_CREATE_CALLOUT
#if KERNEL_THREAD_EXIT_CALLOUT
//---------------------------------------------------------------------------
thread_exit_callout_t Kernel_GetThreadExitCallout(void)
{
    return (thread_exit_callout_t)Kernel::GetThreadExitCallout();
}
#endif // #if KERNEL_THREAD_EXIT_CALLOUT
#if KERNEL_CONTEXT_SWITCH_CALLOUT
//---------------------------------------------------------------------------
thread_context_callout_t Kernel_GetThreadContextSwitchCallout(void)
{
    return (thread_context_callout_t)Kernel::GetThreadContextSwitchCallout
();
}
#endif // #if KERNEL_CONTEXT_SWITCH_CALLOUT
//---------------------------------------------------------------------------
void Kernel_SetStackGuardThreshold(uint16_t u16Threshold_)
{
```

```
    Kernel::SetStackGuardThreshold(u16Threshold_);
}

//---------------------------------------------------------------------------
uint16_t Kernel_GetStackGuardThreshold(void)
{
    return Kernel::GetStackGuardThreshold();
}

//---------------------------------------------------------------------------
void Kernel_SetDebugPrintFunction(
kernel_debug_print_t pfPrintFunction_)
{
    Kernel::SetDebugPrintFunction(pfPrintFunction_);
}

//---------------------------------------------------------------------------
void Kernel_DebugPrint(const char* szString_)
{
    Kernel::DebugPrint(szString_);
}

//---------------------------------------------------------------------------
// Scheduler APIs
void Scheduler_Enable(bool bEnable_)
{
    Scheduler::SetScheduler(bEnable_);
}

//---------------------------------------------------------------------------
bool Scheduler_IsEnabled(void)
{
    return Scheduler::IsEnabled();
}

//---------------------------------------------------------------------------
Thread_t Scheduler_GetCurrentThread(void)
{
    return (Thread_t)Scheduler::GetCurrentThread();
}

//---------------------------------------------------------------------------
// Thread APIs
//---------------------------------------------------------------------------

void Thread_Init(Thread_t        handle,
                 K_WORD*         pwStack_,
                 uint16_t        u16StackSize_,
                 PORT_PRIO_TYPE  uXPriority_,
                 ThreadEntryFunc pfEntryPoint_,
                 void*           pvArg_)
{
    Thread* pclThread = new ((void*)handle) Thread();
    pclThread->Init(pwStack_, u16StackSize_, uXPriority_, pfEntryPoint_, pvArg_);
}

//---------------------------------------------------------------------------
void Thread_Start(Thread_t handle)
{
    Thread* pclThread = (Thread*)handle;
    pclThread->Start();
}

//---------------------------------------------------------------------------
void Thread_Stop(Thread_t handle)
{
    Thread* pclThread = (Thread*)handle;
    pclThread->Stop();
}
//---------------------------------------------------------------------------
void Thread_SetName(Thread_t handle, const char* szName_)
{
    Thread* pclThread = (Thread*)handle;
    pclThread->SetName(szName_);
}
//---------------------------------------------------------------------------
const char* Thread_GetName(Thread_t handle)
{
    Thread* pclThread = (Thread*)handle;
    return pclThread->GetName();
}

//---------------------------------------------------------------------------
PORT_PRIO_TYPE Thread_GetPriority(Thread_t handle)
{
    Thread* pclThread = (Thread*)handle;
    return pclThread->GetPriority();
```

```
}
//---------------------------------------------------------------------------
PORT_PRIO_TYPE Thread_GetCurPriority(Thread_t handle)
{
    Thread* pclThread = (Thread*)handle;
    return pclThread->GetCurPriority();
}

//---------------------------------------------------------------------------
void Thread_SetQuantum(Thread_t handle, uint16_t u16Quantum_)
{
    Thread* pclThread = (Thread*)handle;
    pclThread->SetQuantum(u16Quantum_);
}
//---------------------------------------------------------------------------
uint16_t Thread_GetQuantum(Thread_t handle)
{
    Thread* pclThread = (Thread*)handle;
    return pclThread->GetQuantum();
}

//---------------------------------------------------------------------------
void Thread_SetPriority(Thread_t handle,
PORT_PRIO_TYPE uXPriority_)
{
    Thread* pclThread = (Thread*)handle;
    pclThread->SetPriority(uXPriority_);
}

//---------------------------------------------------------------------------
void Thread_Exit(Thread_t handle)
{
    Thread* pclThread = (Thread*)handle;
    pclThread->Exit();
}

//---------------------------------------------------------------------------
void Thread_Sleep(uint32_t u32TimeMs_)
{
    Thread::Sleep(u32TimeMs_);
}

#if KERNEL_EXTENDED_CONTEXT
//---------------------------------------------------------------------------
void* Thread_GetExtendedContext(Thread_t handle)
{
    Thread* pclThread = (Thread*)handle;
    return pclThread->GetExtendedContext();
}

//---------------------------------------------------------------------------
void Thread_SetExtendedContext(Thread_t handle, void* pvData_)
{
    Thread* pclThread = (Thread*)handle;
    pclThread->SetExtendedContext(pvData_);
}
#endif // #if KERNEL_EXTENDED_CONTEXT

//---------------------------------------------------------------------------
void Thread_Yield(void)
{
    Thread::Yield();
}
//---------------------------------------------------------------------------
void Thread_CoopYield(void)
{
    Thread::CoopYield();
}
//---------------------------------------------------------------------------
void Thread_SetID(Thread_t handle, uint8_t u8ID_)
{
    Thread* pclThread = (Thread*)handle;
    pclThread->SetID(u8ID_);
}
//---------------------------------------------------------------------------
uint8_t Thread_GetID(Thread_t handle)
{
    Thread* pclThread = (Thread*)handle;
    return pclThread->GetID();
}
//---------------------------------------------------------------------------
uint16_t Thread_GetStackSlack(Thread_t handle)
{
    Thread* pclThread = (Thread*)handle;
    return pclThread->GetStackSlack();
}
//---------------------------------------------------------------------------
```

```
thread_state_t Thread_GetState(Thread_t handle)
{
    Thread* pclThread = (Thread*)handle;
    return (thread_state_t)pclThread->GetState();
}
//---------------------------------------------------------------------------
// Timer APIs
//---------------------------------------------------------------------------

//---------------------------------------------------------------------------
void Timer_Init(Timer_t handle)
{
    Timer* pclTimer = new ((void*)handle) Timer();
    pclTimer->Init();
}

//---------------------------------------------------------------------------
void Timer_Start(Timer_t handle, bool bRepeat_, uint32_t u32IntervalMs_,
timer_callback_t pfCallback_, void* pvData_)
{
    Timer* pclTimer = (Timer*)handle;
    pclTimer->Start(bRepeat_, u32IntervalMs_, (TimerCallback)pfCallback_, pvData_);
}

//---------------------------------------------------------------------------
void Timer_Stop(Timer_t handle)
{
    Timer* pclTimer = (Timer*)handle;
    pclTimer->Stop();
}

//---------------------------------------------------------------------------
void Timer_Restart(Timer_t handle)
{
    Timer* pclTimer = (Timer*)handle;
    pclTimer->Start();
}

//---------------------------------------------------------------------------
// Semaphore APIs
//---------------------------------------------------------------------------
void Semaphore_Init(Semaphore_t handle, uint16_t u16InitVal_, uint16_t u16MaxVal_)
{
    Semaphore* pclSemaphore = new ((void*)handle) Semaphore();
    pclSemaphore->Init(u16InitVal_, u16MaxVal_);
}

//---------------------------------------------------------------------------
void Semaphore_Post(Semaphore_t handle)
{
    Semaphore* pclSemaphore = (Semaphore*)handle;
    pclSemaphore->Post();
}

//---------------------------------------------------------------------------
void Semaphore_Pend(Semaphore_t handle)
{
    Semaphore* pclSemaphore = (Semaphore*)handle;
    pclSemaphore->Pend();
}

//---------------------------------------------------------------------------
bool Semaphore_TimedPend(Semaphore_t handle, uint32_t u32WaitTimeMS_)
{
    Semaphore* pclSemaphore = (Semaphore*)handle;
    return pclSemaphore->Pend(u32WaitTimeMS_);
}

//---------------------------------------------------------------------------
// Mutex APIs
//---------------------------------------------------------------------------
void Mutex_Init(Mutex_t handle)
{
    Mutex* pclMutex = new ((void*)handle) Mutex();
    pclMutex->Init();
}

//---------------------------------------------------------------------------
void Mutex_Claim(Mutex_t handle)
{
    Mutex* pclMutex = (Mutex*)handle;
    pclMutex->Claim();
}

//---------------------------------------------------------------------------
void Mutex_Release(Mutex_t handle)
{
```

```
    Mutex* pclMutex = (Mutex*)handle;
    pclMutex->Release();
}

//---------------------------------------------------------------------------
bool Mutex_TimedClaim(Mutex_t handle, uint32_t u32WaitTimeMS_)
{
    Mutex* pclMutex = (Mutex*)handle;
    return pclMutex->Claim(u32WaitTimeMS_);
}

#if KERNEL_EVENT_FLAGS
//---------------------------------------------------------------------------
// EventFlag APIs
//---------------------------------------------------------------------------
void EventFlag_Init(EventFlag_t handle)
{
    EventFlag* pclFlag = new ((void*)handle) EventFlag();
    pclFlag->Init();
}

//---------------------------------------------------------------------------
uint16_t EventFlag_Wait(EventFlag_t handle, uint16_t u16Mask_, event_flag_operation_t eMode_)
{
    EventFlag* pclFlag = (EventFlag*)handle;
    return pclFlag->Wait(u16Mask_, (EventFlagOperation)eMode_);
}

//---------------------------------------------------------------------------
uint16_t EventFlag_TimedWait(EventFlag_t handle, uint16_t u16Mask_, event_flag_operation_t eMode_, uint32_t
 u32TimeMS_)
{
    EventFlag* pclFlag = (EventFlag*)handle;
    return pclFlag->Wait(u16Mask_, (EventFlagOperation)eMode_, u32TimeMS_);
}

//---------------------------------------------------------------------------
void EventFlag_Set(EventFlag_t handle, uint16_t u16Mask_)
{
    EventFlag* pclFlag = (EventFlag*)handle;
    pclFlag->Set(u16Mask_);
}

//---------------------------------------------------------------------------
void EventFlag_Clear(EventFlag_t handle, uint16_t u16Mask_)
{
    EventFlag* pclFlag = (EventFlag*)handle;
    pclFlag->Clear(u16Mask_);
}

//---------------------------------------------------------------------------
uint16_t EventFlag_GetMask(EventFlag_t handle)
{
    EventFlag* pclFlag = (EventFlag*)handle;
    return pclFlag->GetMask();
}
#endif // #if #if KERNEL_EVENT_FLAGS

//---------------------------------------------------------------------------
// Notification APIs
//---------------------------------------------------------------------------
void Notify_Init(Notify_t handle)
{
    Notify* pclNotify = new ((void*)handle) Notify();
    pclNotify->Init();
}

//---------------------------------------------------------------------------
void Notify_Signal(Notify_t handle)
{
    Notify* pclNotify = (Notify*)handle;
    pclNotify->Signal();
}

//---------------------------------------------------------------------------
void Notify_Wait(Notify_t handle, bool* pbFlag_)
{
    Notify* pclNotify = (Notify*)handle;
    pclNotify->Wait(pbFlag_);
}

//---------------------------------------------------------------------------
bool Notify_TimedWait(Notify_t handle, uint32_t u32WaitTimeMS_, bool* pbFlag_)
{
    Notify* pclNotify = (Notify*)handle;
    return pclNotify->Wait(u32WaitTimeMS_, pbFlag_);
}
```

```
//---------------------------------------------------------------------------
// Atomic Functions
//---------------------------------------------------------------------------
uint8_t Atomic_Set8(uint8_t* pu8Source_, uint8_t u8Val_)
{
    return Atomic::Set(pu8Source_, u8Val_);
}

//---------------------------------------------------------------------------
uint16_t Atomic_Set16(uint16_t* pu16Source_, uint16_t u16Val_)
{
    return Atomic::Set(pu16Source_, u16Val_);
}

//---------------------------------------------------------------------------
uint32_t Atomic_Set32(uint32_t* pu32Source_, uint32_t u32Val_)
{
    return Atomic::Set(pu32Source_, u32Val_);
}

//---------------------------------------------------------------------------
uint8_t Atomic_Add8(uint8_t* pu8Source_, uint8_t u8Val_)
{
    return Atomic::Add(pu8Source_, u8Val_);
}

//---------------------------------------------------------------------------
uint16_t Atomic_Add16(uint16_t* pu16Source_, uint16_t u16Val_)
{
    return Atomic::Add(pu16Source_, u16Val_);
}

//---------------------------------------------------------------------------
uint32_t Atomic_Add32(uint32_t* pu32Source_, uint32_t u32Val_)
{
    return Atomic::Add(pu32Source_, u32Val_);
}

//---------------------------------------------------------------------------
uint8_t Atomic_Sub8(uint8_t* pu8Source_, uint8_t u8Val_)
{
    return Atomic::Sub(pu8Source_, u8Val_);
}

//---------------------------------------------------------------------------
uint16_t Atomic_Sub16(uint16_t* pu16Source_, uint16_t u16Val_)
{
    return Atomic::Sub(pu16Source_, u16Val_);
}

//---------------------------------------------------------------------------
uint32_t Atomic_Sub32(uint32_t* pu32Source_, uint32_t u32Val_)
{
    return Atomic::Sub(pu32Source_, u32Val_);
}

//---------------------------------------------------------------------------
bool Atomic_TestAndSet(bool* pbLock)
{
    return Atomic::TestAndSet(pbLock);
}

//---------------------------------------------------------------------------
// Message/Message Queue APIs
//---------------------------------------------------------------------------
void Message_Init(Message_t handle)
{
    Message* pclMessage = new ((void*)handle) Message();
    return pclMessage->Init();
}

//---------------------------------------------------------------------------
void Message_SetData(Message_t handle, void* pvData_)
{
    Message* pclMessage = (Message*)handle;
    pclMessage->SetData(pvData_);
}

//---------------------------------------------------------------------------
void* Message_GetData(Message_t handle)
{
    Message* pclMessage = (Message*)handle;
    return pclMessage->GetData();
}

//---------------------------------------------------------------------------
```

```
void Message_SetCode(Message_t handle, uint16_t u16Code_)
{
    Message* pclMessage = (Message*)handle;
    pclMessage->SetCode(u16Code_);
}

//---------------------------------------------------------------------------
uint16_t Message_GetCode(Message_t handle)
{
    Message* pclMessage = (Message*)handle;
    return pclMessage->GetCode();
}

//---------------------------------------------------------------------------
void MessageQueue_Init(MessageQueue_t handle)
{
    MessageQueue* pclMsgQ = new ((void*)handle) MessageQueue();
    pclMsgQ->Init();
}

//---------------------------------------------------------------------------
Message_t MessageQueue_Receive(MessageQueue_t handle)
{
    MessageQueue* pclMsgQ = (MessageQueue*)handle;
    return pclMsgQ->Receive();
}

//---------------------------------------------------------------------------
void MessagePool_Init(MessagePool_t handle)
{
    MessagePool* pclMsgPool = new ((void*)handle) MessagePool();
    pclMsgPool->Init();
}

//---------------------------------------------------------------------------
void MessagePool_Push(MessagePool_t handle,
      Message_t msg)
{
    MessagePool* pclMsgPool = (MessagePool*)handle;
    pclMsgPool->Push((Message*)msg);
}

//---------------------------------------------------------------------------
Message_t MessagePool_Pop(MessagePool_t handle)
{
    MessagePool* pclMsgPool = (MessagePool*)handle;
    return (Message_t)pclMsgPool->Pop();
}

//---------------------------------------------------------------------------
Message_t MessageQueue_TimedReceive(
      MessageQueue_t handle, uint32_t u32TimeWaitMS_)
{
    MessageQueue* pclMsgQ = (MessageQueue*)handle;
    return (Message_t)pclMsgQ->Receive(u32TimeWaitMS_);
}

//---------------------------------------------------------------------------
void MessageQueue_Send(MessageQueue_t handle,
      Message_t hMessage_)
{
    MessageQueue* pclMsgQ = (MessageQueue*)handle;
    pclMsgQ->Send((Message*)hMessage_);
}

//---------------------------------------------------------------------------
uint16_t MessageQueue_GetCount(MessageQueue_t handle)
{
    MessageQueue* pclMsgQ = (MessageQueue*)handle;
    return pclMsgQ->GetCount();
}

//---------------------------------------------------------------------------
// Mailbox APIs
//---------------------------------------------------------------------------
void Mailbox_Init(Mailbox_t handle, void* pvBuffer_, uint16_t u16BufferSize_, uint16_t
       u16ElementSize_)
{
    Mailbox* pclMBox = new ((void*)handle) Mailbox();
    pclMBox->Init(pvBuffer_, u16BufferSize_, u16ElementSize_);
}

//---------------------------------------------------------------------------
bool Mailbox_Send(Mailbox_t handle, void* pvData_)
{
    Mailbox* pclMBox = (Mailbox*)handle;
    return pclMBox->Send(pvData_);
```

```
}

//---------------------------------------------------------------------------
bool Mailbox_SendTail(Mailbox_t handle, void* pvData_)
{
    Mailbox* pclMBox = (Mailbox*)handle;
    return pclMBox->SendTail(pvData_);
}

//---------------------------------------------------------------------------
bool Mailbox_TimedSend(Mailbox_t handle, void* pvData_, uint32_t u32TimeoutMS_)
{
    Mailbox* pclMBox = (Mailbox*)handle;
    return pclMBox->Send(pvData_, u32TimeoutMS_);
}

//---------------------------------------------------------------------------
bool Mailbox_TimedSendTail(Mailbox_t handle, void* pvData_, uint32_t
u32TimeoutMS_)
{
    Mailbox* pclMBox = (Mailbox*)handle;
    return pclMBox->SendTail(pvData_, u32TimeoutMS_);
}

//---------------------------------------------------------------------------
void Mailbox_Receive(Mailbox_t handle, void* pvData_)
{
    Mailbox* pclMBox = (Mailbox*)handle;
    pclMBox->Receive(pvData_);
}

//---------------------------------------------------------------------------
void Mailbox_ReceiveTail(Mailbox_t handle, void* pvData_)
{
    Mailbox* pclMBox = (Mailbox*)handle;
    pclMBox->ReceiveTail(pvData_);
}

//---------------------------------------------------------------------------
bool Mailbox_TimedReceive(Mailbox_t handle, void* pvData_, uint32_t
u32TimeoutMS_)
{
    Mailbox* pclMBox = (Mailbox*)handle;
    return pclMBox->Receive(pvData_, u32TimeoutMS_);
}

//---------------------------------------------------------------------------
bool Mailbox_TimedReceiveTail(Mailbox_t handle, void* pvData_, uint32_t
u32TimeoutMS_)
{
    Mailbox* pclMBox = (Mailbox*)handle;
    return pclMBox->ReceiveTail(pvData_, u32TimeoutMS_);
}

//---------------------------------------------------------------------------
uint16_t Mailbox_GetFreeSlots(Mailbox_t handle)
{
    Mailbox* pclMBox = (Mailbox*)handle;
    return pclMBox->GetFreeSlots();
}

//---------------------------------------------------------------------------
bool Mailbox_IsFull(Mailbox_t handle)
{
    Mailbox* pclMBox = (Mailbox*)handle;
    return pclMBox->IsFull();
}

//---------------------------------------------------------------------------
bool Mailbox_IsEmpty(Mailbox_t handle)
{
    Mailbox* pclMBox = (Mailbox*)handle;
    return pclMBox->IsEmpty();
}

//---------------------------------------------------------------------------
// Condition Variables
//---------------------------------------------------------------------------
void ConditionVariable_Init(ConditionVariable_t handle)
{
    ConditionVariable* pclCondvar = new ((void*)handle)
ConditionVariable();
    pclCondvar->Init();
}
//---------------------------------------------------------------------------
void ConditionVariable_Wait(ConditionVariable_t handle,
Mutex_t hMutex_)
```

```
{
    ConditionVariable* pclCondvar = (ConditionVariable*)handle;
    Mutex*             pclMutex   = (Mutex*)hMutex_;
    pclCondvar->Wait(pclMutex);
}
//---------------------------------------------------------------------------
void ConditionVariable_Signal(ConditionVariable_t handle)
{
    ConditionVariable* pclCondvar = (ConditionVariable*)handle;
    pclCondvar->Signal();
}
//---------------------------------------------------------------------------
void ConditionVariable_Broadcast(ConditionVariable_t handle)
{
    ConditionVariable* pclCondvar = (ConditionVariable*)handle;
    pclCondvar->Broadcast();
}
//---------------------------------------------------------------------------
bool ConditionVariable_TimedWait(ConditionVariable_t handle, Mutex_t hMutex_, uint32_t u32WaitTimeMS_)
{
    ConditionVariable* pclCondvar = (ConditionVariable*)handle;
    Mutex*             pclMutex   = (Mutex*)hMutex_;
    return pclCondvar->Wait(pclMutex, u32WaitTimeMS_);
}

//---------------------------------------------------------------------------
// Reader-writer locks
//---------------------------------------------------------------------------
void ReaderWriterLock_Init(ReaderWriterLock_t handle)
{
    ReaderWriterLock* pclReaderWriter = new ((void*)handle) ReaderWriterLock();
    pclReaderWriter->Init();
}

//---------------------------------------------------------------------------
void ReaderWriterLock_AcquireReader(ReaderWriterLock_t handle)
{
    ReaderWriterLock* pclReaderWriter = (ReaderWriterLock*)handle;
    pclReaderWriter->AcquireReader();
}

//---------------------------------------------------------------------------
void ReaderWriterLock_ReleaseReader(ReaderWriterLock_t handle)
{
    ReaderWriterLock* pclReaderWriter = (ReaderWriterLock*)handle;
    pclReaderWriter->ReleaseReader();
}

//---------------------------------------------------------------------------
void ReaderWriterLock_AcquireWriter(ReaderWriterLock_t handle)
{
    ReaderWriterLock* pclReaderWriter = (ReaderWriterLock*)handle;
    pclReaderWriter->AcquireWriter();
}

//---------------------------------------------------------------------------
void ReaderWriterLock_ReleaseWriter(ReaderWriterLock_t handle)
{
    ReaderWriterLock* pclReaderWriter = (ReaderWriterLock*)handle;
    pclReaderWriter->ReleaseWriter();
}

//---------------------------------------------------------------------------
bool ReaderWriterLock_TimedAcquireWriter(ReaderWriterLock_t handle, uint32_t u32TimeoutMs_)
{
    ReaderWriterLock* pclReaderWriter = (ReaderWriterLock*)handle;
    return pclReaderWriter->AcquireWriter(u32TimeoutMs_);
}

//---------------------------------------------------------------------------
bool ReaderWriterLock_TimedAcquireReader(ReaderWriterLock_t handle, uint32_t u32TimeoutMs_)
{
    ReaderWriterLock* pclReaderWriter = (ReaderWriterLock*)handle;
    return pclReaderWriter->AcquireReader(u32TimeoutMs_);
}
```

20.3 /home/moslevin/projects/m3-repo/kernel/lib/mark3c/src/public/fake_types.h File Reference

C-struct definitions that mirror.

```
#include <stdint.h>
#include <stddef.h>
#include <stdbool.h>
#include "mark3cfg.h"
```

20.3.1 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.

20.4 fake_types.h

```
/*===========================================================================
     _____     _____     _____     _____
 ___|    _|__ __|_    |__ __|__   |__ __| __  |__ _____
|    \  /  | ||    \      ||     |     ||  |/ /     ||___  |
|     \/   | ||     \     ||     \     ||     \     ||___  |
|__/\__/|__|_||__|\__\  __||__|\__\  __||__|\__\  __||_____|
    |_____|      |_____|      |_____|      |_____|

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

Copyright (c) 2012 - 2019 m0slevin, all rights reserved.
See license.txt for more information
===========================================================================*/
#include <stdint.h>
#include <stddef.h>
#include <stdbool.h>
#include "mark3cfg.h"

#pragma once
#if defined(__cplusplus)
extern "C" {
#endif

//---------------------------------------------------------------------------
typedef struct {
    void* prev;
    void* next;
} Fake_LinkedListNode;

//---------------------------------------------------------------------------
typedef struct {
    void* head;
    void* tail;
} Fake_LinkedList;

//---------------------------------------------------------------------------
typedef struct {
    Fake_LinkedListNode fake_node;
    Fake_LinkedList fake_list;
    PORT_PRIO_TYPE  m_uXPriority;
    void*           m_pclMap;
} Fake_ThreadList;

//---------------------------------------------------------------------------
```

```
typedef struct {
    Fake_LinkedListNode m_ll_node;
    uint8_t             m_u8Initialized;
    uint8_t             m_u8Flags;
    void*               m_pfCallback;
    uint32_t            m_u32Interval;
    uint32_t            m_u32TimeLeft;
    void*               m_pclOwner;
    void*               m_pvData;
} Fake_Timer;

//---------------------------------------------------------------------------
typedef struct {
    Fake_LinkedListNode m_ll_node;
    K_WORD*             m_pwStackTop;
    K_WORD*             m_pwStack;
    uint8_t             m_u8ThreadID;
    PORT_PRIO_TYPE      m_uXPriority;
    PORT_PRIO_TYPE      m_uXCurPriority;
    uint8_t             m_eState;
#if KERNEL_EXTENDED_CONTEXT
    void* m_pvExtendedContext;
#endif // #if KERNEL_EXTENDED_CONTEXT
#if KERNEL_NAMED_THREADS
    const char* m_szName;
#endif // #if KERNEL_NAMED_THREADS
    uint16_t m_u16StackSize;
    void*    m_pclCurrent;
    void*    m_pclOwner;
    void*    m_pfEntryPoint;
    void*    m_pvArg;
#if KERNEL_ROUND_ROBIN
    uint16_t m_u16Quantum;
#endif // #if KERNEL_ROUND_ROBIN
#if KERNEL_EVENT_FLAGS
    uint16_t m_u16FlagMask;
    uint8_t  m_eFlagMode;
#endif // #if KERNEL_EVENT_FLAGS
    Fake_Timer m_clTimer;
    bool       m_bExpired;
} Fake_Thread;

//---------------------------------------------------------------------------
typedef struct {
    Fake_ThreadList thread_list;
    uint8_t         m_u8Initialized;
    uint16_t        m_u16Value;
    uint16_t        m_u16MaxValue;
} Fake_Semaphore;

//---------------------------------------------------------------------------
typedef struct {
    Fake_ThreadList thread_list;
    uint8_t         m_u8Initialized;
    uint8_t         m_u8Recurse;
    bool            m_bReady;
    bool            m_bRecursive;
    uint8_t         m_u8MaxPri;
    void*           m_pclOwner;
} Fake_Mutex;

//---------------------------------------------------------------------------
typedef struct {
    Fake_LinkedListNode list_node;
    void*               m_pvData;
    uint16_t            m_u16Code;
} Fake_Message;

//---------------------------------------------------------------------------
typedef struct {
    Fake_Semaphore  m_clSemaphore;
    Fake_LinkedList m_clLinkList;
} Fake_MessageQueue;

//---------------------------------------------------------------------------
typedef struct {
    Fake_LinkedList m_clList;
} Fake_MessagePool;

//---------------------------------------------------------------------------
typedef struct {
    uint16_t       m_u16Head;
    uint16_t       m_u16Tail;
    uint16_t       m_u16Count;
    uint16_t       m_u16Free;
    uint16_t       m_u16ElementSize;
    void*          m_pvBuffer;
```

```
    Fake_Semaphore m_clRecvSem;
    Fake_Semaphore m_clSendSem;
} Fake_Mailbox;

//---------------------------------------------------------------------------
typedef struct {
    Fake_ThreadList thread_list;
    uint8_t         m_u8Initialized;
    bool            m_bPending;
} Fake_Notify;

//---------------------------------------------------------------------------
typedef struct {
    Fake_ThreadList thread_list;
    uint8_t         m_u8Initialized;
    uint16_t        m_u16EventFlag;
} Fake_EventFlag;

//---------------------------------------------------------------------------
typedef struct {
    Fake_Mutex m_clGlobalMutex;
    Fake_Mutex m_clReaderMutex;
    uint8_t    m_u8ReadCount;
} Fake_ReaderWriterLock;

//---------------------------------------------------------------------------
typedef struct {
    Fake_Mutex     m_clMutex;
    Fake_Semaphore m_clSemaphore;
    uint8_t        m_u8Waiters;
} Fake_ConditionVariable;

#if defined(__cplusplus)
}
#endif

```

20.5 /home/moslevin/projects/m3-repo/kernel/lib/mark3c/src/public/mark3c.h File Reference

Header providing C-language API bindings for the Mark3 kernel.

```
#include "mark3cfg.h"
#include "fake_types.h"
#include <stdint.h>
#include <stdbool.h>
```

Macros

- #define THREAD_SIZE (sizeof(Fake_Thread))
- #define TIMER_SIZE (sizeof(Fake_Timer))
- #define SEMAPHORE_SIZE (sizeof(Fake_Semaphore))
- #define MUTEX_SIZE (sizeof(Fake_Mutex))
- #define MESSAGE_SIZE (sizeof(Fake_Message))
- #define MESSAGEQUEUE_SIZE (sizeof(Fake_MessageQueue))
- #define MAILBOX_SIZE (sizeof(Fake_Mailbox))
- #define NOTIFY_SIZE (sizeof(Fake_Notify))
- #define MESSAGEPOOL_SIZE (sizeof(Fake_MessagePool))
- #define CONDITIONVARIABLE_SIZE (sizeof(Fake_ConditionVariable))
- #define READERWRITERLOCK_SIZE (sizeof(Fake_ReaderWriterLock))
- #define TOKEN_1(x, y) x##y
- #define TOKEN_2(x, y) TOKEN_1(x, y)
- #define WORD_ROUND(x) (((x) + (sizeof(K_WORD) - 1)) / sizeof(K_WORD))

- #define DECLARE_THREAD(name)
- #define DECLARE_TIMER(name)
- #define DECLARE_SEMAPHORE(name)
- #define DECLARE_MUTEX(name)
- #define DECLARE_MESSAGE(name)
- #define DECLARE_MESSAGEPOOL(name)
- #define DECLARE_MESSAGEQUEUE(name)
- #define DECLARE_MAILBOX(name)
- #define DECLARE_NOTIFY(name)
- #define DECLARE_CONDITIONVARIABLE(name)
- #define DECLARE_READERWRITERLOCK(name)

Typedefs

- typedef void ∗ Mailbox_t

    Mailbox opaque handle data type.
- typedef void ∗ Message_t

    Message opaque handle data type.
- typedef void ∗ MessagePool_t

    MessagePool 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.
- typedef void ∗ ConditionVariable_t

    Condition Variable opaque handle data type.
- typedef void ∗ ReaderWriterLock_t

    Reader-writer-lock opaque handle data type.
- typedef void(∗ kernel_debug_print_t) (const char ∗szString_)
- typedef void(∗ panic_func_t) (uint16_t u16PanicCode_)
- typedef void(∗ thread_entry_func_t) (void ∗pvArg_)
- typedef void(∗ timer_callback_t) (Thread_t hOwner_, void ∗pvData_)

Enumerations

- enum thread_state_t {
  THREAD_STATE_EXIT = 0, THREAD_STATE_READY, THREAD_STATE_BLOCKED, THREAD_STATE↩
  _STOP,
  THREAD_STATE_INVALID }

Functions

- void * Alloc_Memory (size_t eSize_)

  Alloc_Memory.
- void Free_Memory (void *pvObject_)

  Free_Memory.
- Semaphore_t Alloc_Semaphore (void)

  Alloc_Semaphore.
- void Free_Semaphore (Semaphore_t handle)
- Mutex_t Alloc_Mutex (void)

  Alloc_Mutex.
- void Free_Mutex (Mutex_t handle)
- Message_t Alloc_Message (void)

  Alloc_Message.
- void Free_Message (Message_t handle)
- MessageQueue_t Alloc_MessageQueue (void)

  Alloc_MessageQueue.
- void Free_MessageQueue (MessageQueue_t handle)
- MessagePool_t Alloc_MessagePool (void)
- void Free_MessagePool (MessagePool_t handle)
- Notify_t Alloc_Notify (void)

  Alloc_Notify.
- void Free_Notify (Notify_t handle)
- Mailbox_t Alloc_Mailbox (void)

  Alloc_Mailbox.
- void Free_Mailbox (Mailbox_t handle)
- Thread_t Alloc_Thread (void)

  Alloc_Thread.
- void Free_Thread (Thread_t handle)
- Timer_t Alloc_Timer (void)

  Alloc_Timer.
- void Free_Timer (Timer_t handle)
- void Kernel_Init (void)

  Kernel_Init.
- void Kernel_Start (void)

  Kernel_Start.
- bool Kernel_IsStarted (void)

  Kernel_IsStarted.
- void Kernel_SetPanic (panic_func_t pfPanic_)

  Kernel_SetPanic.
- bool Kernel_IsPanic (void)

  Kernel_IsPanic.
- void Kernel_Panic (uint16_t u16Cause_)

  Kernel_Panic.
- uint32_t Kernel_GetTicks (void)

  Kernel_GetTicks.
- 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_, thread_entry_func_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_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_Yield (void)

  Thread_Yield.
- void Thread_CoopYield (void)

  Thread_CoopYield.
- void Thread_SetID (Thread_t handle, uint8_t u8ID_)

  Thread_SetID.
- uint8_t Thread_GetID (Thread_t handle)

  Thread_GetID.
- thread_state_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_, timer_callback_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 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.
- uint8_t Atomic_Set8 (uint8_t ∗pu8Source_, uint8_t u8Val_)

  Atomic_Set8.
- uint16_t Atomic_Set16 (uint16_t ∗pu16Source_, uint16_t u16Val_)

  Atomic_Set16.
- uint32_t Atomic_Set32 (uint32_t ∗pu32Source_, uint32_t u32Val_)

  Atomic_Set32.
- uint8_t Atomic_Add8 (uint8_t ∗pu8Source_, uint8_t u8Val_)

  Atomic_Add8.
- uint16_t Atomic_Add16 (uint16_t ∗pu16Source_, uint16_t u16Val_)

  Atomic_Add16.
- uint32_t Atomic_Add32 (uint32_t ∗pu32Source_, uint32_t u32Val_)

  Atomic_Add32.
- uint8_t Atomic_Sub8 (uint8_t ∗pu8Source_, uint8_t u8Val_)

  Atomic_Sub8.
- uint16_t Atomic_Sub16 (uint16_t ∗pu16Source_, uint16_t u16Val_)

  Atomic_Sub16.
- uint32_t Atomic_Sub32 (uint32_t ∗pu32Source_, uint32_t u32Val_)

  Atomic_Sub32.
- bool Atomic_TestAndSet (bool ∗pbLock)

  Atomic_TestAndSet.
- 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 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 (MessageQueue_t handle)

  MessageQueue_GetCount.
- void MessagePool_Init (MessagePool_t handle)

  MessagePool_Init.

- void MessagePool_Push (MessagePool_t handle, Message_t msg)

    MessagePool_Push.
- Message_t MessagePool_Pop (MessagePool_t handle)

    MessagePool_Pop.
- 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 ConditionVariable_Init (ConditionVariable_t handle)

    ConditionVariable_Init.
- void ConditionVariable_Wait (ConditionVariable_t handle, Mutex_t hMutex_)

    ConditionVariable_Wait.
- void ConditionVariable_Signal (ConditionVariable_t handle)

    ConditionVariable_Signal.
- void ConditionVariable_Broadcast (ConditionVariable_t handle)

    ConditionVariable_Broadcast.
- bool ConditionVariable_TimedWait (ConditionVariable_t handle, Mutex_t hMutex_, uint32_t u32WaitTime↩ MS_)

    ConditionVariable_TimedWait.
- void ReaderWriterLock_Init (ReaderWriterLock_t handle)

    ReaderWriterLock_Init.
- void ReaderWriterLock_AcquireReader (ReaderWriterLock_t handle)

    ReaderWriterLock_AcquireReader.
- void ReaderWriterLock_ReleaseReader (ReaderWriterLock_t handle)

    ReaderWriterLock_ReleaseReader.
- void ReaderWriterLock_AcquireWriter (ReaderWriterLock_t handle)

    ReaderWriterLock_AcquireWriter.
- void ReaderWriterLock_ReleaseWriter (ReaderWriterLock_t handle)

    ReaderWriterLock_ReleaseWriter.
- bool ReaderWriterLock_TimedAcquireWriter (ReaderWriterLock_t handle, uint32_t u32TimeoutMs_)

    ReaderWriterLock_TimedAcquireWriter.

- bool ReaderWriterLock_TimedAcquireReader (ReaderWriterLock_t handle, uint32_t u32TimeoutMs_)

  ReaderWriterLock_TimedAcquireReader.
- void Kernel_SetDebugPrintFunction (kernel_debug_print_t pfPrintFunction_)

  Kernel_SetDebugPrintFunction.
- void Kernel_DebugPrint (const char *szString_)

  KernelDebug_DebugPrint.

20.5.1 Detailed Description

Header providing C-language API bindings for the Mark3 kernel.

Definition in file mark3c.h.

20.5.2 Macro Definition Documentation

20.5.2.1 CONDITIONVARIABLE_SIZE

```
#define CONDITIONVARIABLE_SIZE (sizeof(Fake_ConditionVariable))
```

Definition at line 79 of file mark3c.h.

20.5.2.2 DECLARE_CONDITIONVARIABLE

```
#define DECLARE_CONDITIONVARIABLE(
                name )
```

Value:

```
K_WORD              TOKEN_2(__condvar_, name)[WORD_ROUND(EVENTFLAG_SIZE)];
                                           \
    ConditionVariable_t name = (ConditionVariable_t)TOKEN_2(__condvar_, name);
```

Definition at line 161 of file mark3c.h.

20.5.2.3 DECLARE_MAILBOX

```
#define DECLARE_MAILBOX(
                name )
```

Value:

```
K_WORD    TOKEN_2(__mailbox_, name)[WORD_ROUND(
      MAILBOX_SIZE)];                                                        \
    Mailbox_t name = (Mailbox_t)TOKEN_2(__mailbox_, name);
```

Definition at line 147 of file mark3c.h.

20.5.2.4 DECLARE_MESSAGE

```
#define DECLARE_MESSAGE(
                name )
```

Value:

```
K_WORD    TOKEN_2(__message_, name)[WORD_ROUND(
      MESSAGE_SIZE)];                                                     \
    Message_t name = (Message_t)TOKEN_2(__message_, name);
```

Definition at line 135 of file mark3c.h.

20.5.2.5 DECLARE_MESSAGEPOOL

```
#define DECLARE_MESSAGEPOOL(
                name )
```

Value:

```
K_WORD        TOKEN_2(__messagepool_, name)[WORD_ROUND(
      MESSAGEPOOL_SIZE)];                                         \
    MessagePool_t name = (MessagePool_t)TOKEN_2(__messagepool_, name);
```

Definition at line 139 of file mark3c.h.

20.5.2.6 DECLARE_MESSAGEQUEUE

```
#define DECLARE_MESSAGEQUEUE(
                name )
```

Value:

```
K_WORD         TOKEN_2(__messagequeue_, name)[WORD_ROUND(
      MESSAGEQUEUE_SIZE)];                                        \
    MessageQueue_t name = (MessageQueue_t)TOKEN_2(__messagequeue_, name);
```

Definition at line 143 of file mark3c.h.

20.5.2.7 DECLARE_MUTEX

```
#define DECLARE_MUTEX(
              name )
```

Value:

```
K_WORD  TOKEN_2(__mutex_, name)[WORD_ROUND(MUTEX_SIZE)];
                                                 \
    Mutex_t name = (Mutex_t)TOKEN_2(__mutex_, name);
```

Definition at line 131 of file mark3c.h.

20.5.2.8 DECLARE_NOTIFY

```
#define DECLARE_NOTIFY(
              name )
```

Value:

```
K_WORD   TOKEN_2(__notify_, name)[WORD_ROUND(NOTIFY_SIZE)];
                                                   \
    Notify_t name = (Notify_t)TOKEN_2(__notify_, name);
```

Definition at line 151 of file mark3c.h.

20.5.2.9 DECLARE_READERWRITERLOCK

```
#define DECLARE_READERWRITERLOCK(
              name )
```

Value:

```
K_WORD            TOKEN_2(__readerwriterlock_, name)[WORD_ROUND(EVENTFLAG_SIZE)];
                                       \
    ReaderWriterLock_t name = (ReaderWriterLock_t)TOKEN_2(__readerwriterlock_,
      name);
```

Definition at line 165 of file mark3c.h.

20.5.2.10 DECLARE_SEMAPHORE

```
#define DECLARE_SEMAPHORE(
                name )
```

Value:

```
K_WORD      TOKEN_2(__semaphore_, name)[WORD_ROUND(
      SEMAPHORE_SIZE)];                                          \
    Semaphore_t name = (Semaphore_t)TOKEN_2(__semaphore_, name);
```

Definition at line 127 of file mark3c.h.

20.5.2.11 DECLARE_THREAD

```
#define DECLARE_THREAD(
                name )
```

Value:

```
K_WORD   TOKEN_2(__thread_, name)[WORD_ROUND(THREAD_SIZE)];
                                                 \
    Thread_t name = (Thread_t)TOKEN_2(__thread_, name);
```

Definition at line 119 of file mark3c.h.

20.5.2.12 DECLARE_TIMER

```
#define DECLARE_TIMER(
                name )
```

Value:

```
K_WORD  TOKEN_2(__timer_, name)[WORD_ROUND(TIMER_SIZE)];
                                                \
    Timer_t name = (Timer_t)TOKEN_2(__timer_, name);
```

Definition at line 123 of file mark3c.h.

20.5.2.13 MAILBOX_SIZE

```
#define MAILBOX_SIZE (sizeof(Fake_Mailbox))
```

Definition at line 73 of file mark3c.h.

20.5.2.14 MESSAGE_SIZE

```
#define MESSAGE_SIZE (sizeof(Fake_Message))
```

Definition at line 71 of file mark3c.h.

20.5.2.15 MESSAGEPOOL_SIZE

```
#define MESSAGEPOOL_SIZE (sizeof(Fake_MessagePool))
```

Definition at line 78 of file mark3c.h.

20.5.2.16 MESSAGEQUEUE_SIZE

```
#define MESSAGEQUEUE_SIZE (sizeof(Fake_MessageQueue))
```

Definition at line 72 of file mark3c.h.

20.5.2.17 MUTEX_SIZE

```
#define MUTEX_SIZE (sizeof(Fake_Mutex))
```

Definition at line 70 of file mark3c.h.

20.5.2.18 NOTIFY_SIZE

```
#define NOTIFY_SIZE (sizeof(Fake_Notify))
```

Definition at line 74 of file mark3c.h.

20.5.2.19 READERWRITERLOCK_SIZE

```
#define READERWRITERLOCK_SIZE (sizeof(Fake_ReaderWriterLock))
```

Definition at line 80 of file mark3c.h.

20.5.2.20 SEMAPHORE_SIZE

```
#define SEMAPHORE_SIZE (sizeof(Fake_Semaphore))
```

Definition at line 69 of file mark3c.h.

20.5.2.21 THREAD_SIZE

```
#define THREAD_SIZE (sizeof(Fake_Thread))
```

Definition at line 67 of file mark3c.h.

20.5.2.22 TIMER_SIZE

```
#define TIMER_SIZE (sizeof(Fake_Timer))
```

Definition at line 68 of file mark3c.h.

20.5.2.23 TOKEN_1

```
#define TOKEN_1(
              x,
              y ) x##y
```

Definition at line 112 of file mark3c.h.

20.5.2.24 TOKEN_2

```
#define TOKEN_2(
              x,
              y ) TOKEN_1(x, y)
```

Definition at line 113 of file mark3c.h.

20.5.2.25 WORD_ROUND

```
#define WORD_ROUND(
              x ) (((x) + (sizeof(K_WORD) - 1)) / sizeof(K_WORD))
```

Definition at line 117 of file mark3c.h.

20.5.3 Typedef Documentation

20.5.3.1 ConditionVariable_t

```
typedef void* ConditionVariable_t
```

Condition Variable opaque handle data type.

Definition at line 48 of file mark3c.h.

20.5.3.2 kernel_debug_print_t

```
typedef void(* kernel_debug_print_t) (const char *szString_)
```

Definition at line 62 of file mark3c.h.

20.5.3.3 Mailbox_t

```
typedef void* Mailbox_t
```

Mailbox opaque handle data type.

Definition at line 39 of file mark3c.h.

20.5.3.4 Message_t

```
typedef void* Message_t
```

Message opaque handle data type.

Definition at line 40 of file mark3c.h.

20.5.3.5 MessagePool_t

```
typedef void* MessagePool_t
```

MessagePool opaque handle data type.

Definition at line 41 of file mark3c.h.

20.5.3.6 MessageQueue_t

```
typedef void* MessageQueue_t
```

MessageQueue opaque handle data type.

Definition at line 42 of file mark3c.h.

20.5.3.7 Mutex_t

```
typedef void* Mutex_t
```

Mutex opaque handle data type.

Definition at line 43 of file mark3c.h.

20.5.3.8 Notify_t

```
typedef void* Notify_t
```

Notification object opaque handle data type.

Definition at line 44 of file mark3c.h.

20.5.3.9 panic_func_t

```
typedef void(* panic_func_t) (uint16_t u16PanicCode_)
```

Definition at line 282 of file mark3c.h.

20.5.3.10 ReaderWriterLock_t

```
typedef void* ReaderWriterLock_t
```

Reader-writer-lock opaque handle data type.

Definition at line 49 of file mark3c.h.

20.5.3.11 Semaphore_t

```
typedef void* Semaphore_t
```

Semaphore opaque handle data type.

Definition at line 45 of file mark3c.h.

20.5.3.12 thread_entry_func_t

```
typedef void(* thread_entry_func_t) (void *pvArg_)
```

Definition at line 401 of file mark3c.h.

20.5.3.13 Thread_t

```
typedef void* Thread_t
```

Thread opaque handle data type.

Definition at line 46 of file mark3c.h.

20.5.3.14 timer_callback_t

```
typedef void(* timer_callback_t) (Thread_t hOwner_, void *pvData_)
```

Definition at line 570 of file mark3c.h.

20.5.3.15 Timer_t

```
typedef void* Timer_t
```

Timer opaque handle data type.

Definition at line 47 of file mark3c.h.

20.5.4 Enumeration Type Documentation

20.5.4.1 thread_state_t

```
enum thread_state_t
```

Define the various states that a thread can be in

Enumerator

THREAD_STATE_EXIT	
THREAD_STATE_READY	!< Thread has terminated via exit path
THREAD_STATE_BLOCKED	!< Thread is ready to run
THREAD_STATE_STOP	!< Thread is blocked on a blocking call
THREAD_STATE_INVALID	!< Thread has been manually stopped !< Invalid thread state

Definition at line 101 of file mark3c.h.

20.5.5 Function Documentation

20.5.5.1 Alloc_Mailbox()

```
Mailbox_t Alloc_Mailbox (
            void  )
```

Alloc_Mailbox.

See also

Mailbox∗ AutoAlloc::NewMailbox()

Returns

Handle to an allocated object, or nullptr if heap exhausted

Definition at line 123 of file mark3c.cpp.

20.5.5.2 Alloc_Memory()

```
void* Alloc_Memory (
            size_t eSize_ )
```

Alloc_Memory.

See also

void∗ AutoAlloc::NewRawData(size_t sSize_)

Parameters

e← Size← _	Size in bytes to allocate from the one-time-allocate heap

Returns

Pointer to an allocated blob of memory, or nullptr if heap exhausted

20.5.5.3 Alloc_Message()

```
Message_t Alloc_Message (
            void )
```

Alloc_Message.

See also

AutoAlloc::NewMessage()

Returns

Handle to an allocated object, or nullptr if heap exhausted

Definition at line 76 of file mark3c.cpp.

20.5.5.4 Alloc_MessagePool()

```
MessagePool_t Alloc_MessagePool (
            void )
```

Definition at line 100 of file mark3c.cpp.

20.5.5.5 Alloc_MessageQueue()

```
MessageQueue_t Alloc_MessageQueue (
            void )
```

Alloc_MessageQueue.

See also

MesageQueue∗ AutoAlloc::NewMessageQueue()

Returns

Handle to an allocated object, or nullptr if heap exhausted

Definition at line 88 of file mark3c.cpp.

20.5.5.6 Alloc_Mutex()

```
Mutex_t Alloc_Mutex (
            void  )
```

Alloc_Mutex.

See also

Mutex* AutoAlloc::NewMutex()

Returns

Handle to an allocated object, or nullptr if heap exhausted

Definition at line 52 of file mark3c.cpp.

20.5.5.7 Alloc_Notify()

```
Notify_t Alloc_Notify (
            void  )
```

Alloc_Notify.

See also

Notify* AutoAlloc::NewNotify()

Returns

Handle to an allocated object, or nullptr if heap exhausted

Definition at line 112 of file mark3c.cpp.

20.5.5.8 Alloc_Semaphore()

```
Semaphore_t Alloc_Semaphore (
            void  )
```

Alloc_Semaphore.

See also

Semaphore* AutoAlloc::NewSemaphore()

Returns

Handle to an allocated object, or nullptr if heap exhausted

Definition at line 40 of file mark3c.cpp.

20.5.5.9 Alloc_Thread()

```
Thread_t Alloc_Thread (
            void )
```

Alloc_Thread.

See also

Thread∗ AutoAlloc::NewThread()

Returns

Handle to an allocated object, or nullptr if heap exhausted

Definition at line 156 of file mark3c.cpp.

20.5.5.10 Alloc_Timer()

```
Timer_t Alloc_Timer (
            void )
```

Alloc_Timer.

See also

Timer∗ AutoAlloc::NewTimer()

Returns

Handle to an allocated object, or nullptr if heap exhausted

Definition at line 167 of file mark3c.cpp.

20.5.5.11 Atomic_Add16()

```
uint16_t Atomic_Add16 (
            uint16_t * pu16Source_,
            uint16_t u16Val_ )
```

Atomic_Add16.

See also

uint16_t Atomic::Add(uint16_t ∗pu16Source_, uint16_t u16Val_)

Parameters

*pu16←Source_*	Pointer to a variable
*u16Val_*	Value to add to the variable

Returns

Previously-held value in pu16Source_

Definition at line 638 of file mark3c.cpp.

20.5.5.12 Atomic_Add32()

```
uint32_t Atomic_Add32 (
            uint32_t * pu32Source_,
            uint32_t u32Val_ )
```

Atomic_Add32.

See also

uint32_t Atomic::Add(uint32_t ∗pu32Source_, uint32_t u32Val_)

Parameters

*pu32←Source_*	Pointer to a variable
*u32Val_*	Value to add to the variable

Returns

Previously-held value in pu32Source_

Definition at line 644 of file mark3c.cpp.

20.5.5.13 Atomic_Add8()

```
uint8_t Atomic_Add8 (
            uint8_t * pu8Source_,
            uint8_t u8Val_ )
```

Atomic_Add8.

See also

uint8_t Atomic::Add(uint8_t ∗pu8Source_, uint8_t u8Val_)

Parameters

*pu8↔ Source_*	Pointer to a variable
*u8Val_*	Value to add to the variable

Returns

Previously-held value in pu8Source_

Definition at line 632 of file mark3c.cpp.

20.5.5.14 Atomic_Set16()

```
uint16_t Atomic_Set16 (
            uint16_t * pu16Source_,
            uint16_t u16Val_ )
```

Atomic_Set16.

See also

uint16_t Atomic::Set(uint16_t ∗pu16Source_, uint16_t u16Val_)

Parameters

*pu16↔ Source_*	Pointer to a variable to set the value of
*u16Val_*	New value to set in the variable

Returns

Previously-set value

Definition at line 620 of file mark3c.cpp.

20.5.5.15 Atomic_Set32()

```
uint32_t Atomic_Set32 (
            uint32_t * pu32Source_,
            uint32_t u32Val_ )
```

Atomic_Set32.

See also

uint32_t Atomic::Set(uint32_t ∗pu32Source_, uint32_t u32Val_)

Parameters

*pu32↩ Source_*	Pointer to a variable to set the value of
*u32Val_*	New value to set in the variable

Returns

Previously-set value

Definition at line 626 of file mark3c.cpp.

20.5.5.16 Atomic_Set8()

```
uint8_t Atomic_Set8 (
            uint8_t * pu8Source_,
            uint8_t u8Val_ )
```

Atomic_Set8.

See also

uint8_t Atomic::Set(uint8_t ∗pu8Source_, uint8_t u8Val_)

Parameters

*pu8↩ Source_*	Pointer to a variable to set the value of
*u8Val_*	New value to set in the variable

Returns

Previously-set value

Definition at line 614 of file mark3c.cpp.

20.5.5.17 Atomic_Sub16()

```
uint16_t Atomic_Sub16 (
            uint16_t * pu16Source_,
            uint16_t u16Val_ )
```

Atomic_Sub16.

See also

uint16_t Atomic::Sub(uint16_t ∗pu16Source_, uint16_t u16Val_)

Parameters

*pu16←↩ Source_*	Pointer to a variable
*u16Val_*	Value to subtract from the variable

Returns

Previously-held value in pu16Source_

Definition at line 656 of file mark3c.cpp.

20.5.5.18 Atomic_Sub32()

```
uint32_t Atomic_Sub32 (
            uint32_t * pu32Source_,
            uint32_t u32Val_ )
```

Atomic_Sub32.

See also

uint32_t Atomic::Sub(uint32_t ∗pu32Source_, uint32_t u32Val_)

Parameters

*pu32←↩ Source_*	Pointer to a variable
*u32Val_*	Value to subtract from the variable

Returns

Previously-held value in pu32Source_

Definition at line 662 of file mark3c.cpp.

20.5.5.19 Atomic_Sub8()

```
uint8_t Atomic_Sub8 (
            uint8_t * pu8Source_,
            uint8_t u8Val_ )
```

Atomic_Sub8.

See also

uint8_t Atomic::Sub(uint8_t ∗pu8Source_, uint8_t u8Val_)

Parameters

pu8← *Source_*	Pointer to a variable
*u8Val_*	Value to subtract from the variable

Returns

Previously-held value in pu8Source_

Definition at line 650 of file mark3c.cpp.

20.5.5.20 Atomic_TestAndSet()

```
bool Atomic_TestAndSet (
            bool * pbLock )
```

Atomic_TestAndSet.

See also

bool Atomic::TestAndSet(bool *pbLock)

Parameters

pbLock	Pointer to a value to test against. This will always be set to "true" at the end of a call to TestAndSet.

Returns

true - Lock value was "true" on entry, false - Lock was set

Definition at line 668 of file mark3c.cpp.

20.5.5.21 ConditionVariable_Broadcast()

```
void ConditionVariable_Broadcast (
            ConditionVariable_t handle )
```

ConditionVariable_Broadcast.

See also

void ConditionVariable::Broadcast()

Parameters

handle	Handle of the condition variable object

Definition at line 874 of file mark3c.cpp.

20.5.5.22 ConditionVariable_Init()

```
void ConditionVariable_Init (
            ConditionVariable_t handle )
```

ConditionVariable_Init.

See also

void ConditionVariable::Init()

Parameters

handle	Handle of the condition variable object

Definition at line 855 of file mark3c.cpp.

20.5.5.23 ConditionVariable_Signal()

```
void ConditionVariable_Signal (
            ConditionVariable_t handle )
```

ConditionVariable_Signal.

See also

void ConditionVariable::Signal()

Parameters

handle	Handle of the condition variable object

Definition at line 868 of file mark3c.cpp.

20.5.5.24 ConditionVariable_TimedWait()

```
bool ConditionVariable_TimedWait (
            ConditionVariable_t handle,
            Mutex_t hMutex_,
            uint32_t u32WaitTimeMS_ )
```

ConditionVariable_TimedWait.

See also

bool ConditionVariable::Wait(Mutex∗ pclMutex_, uint32_t u32WaitTimeMS_)

Parameters

handle	Handle of the condition variable object
*hMutex_*	Handle of the mutex to lock on acquisition of the condition variable
*u32WaitTimeM↩ S_*	Maximum time to wait for object

Returns

true on success, false on timeout

Definition at line 880 of file mark3c.cpp.

20.5.5.25 ConditionVariable_Wait()

```
void ConditionVariable_Wait (
            ConditionVariable_t handle,
            Mutex_t hMutex_ )
```

ConditionVariable_Wait.

See also

void ConditionVariable::Wait(Mutex∗ pclMutex_)

Parameters

handle	Handle of the condition variable object
h↩ Mutex↩ _	Handle of the mutex to lock on acquisition of the condition variable

Definition at line 861 of file mark3c.cpp.

20.5.5.26 Free_Mailbox()

```
void Free_Mailbox (
            Mailbox_t handle )
```

Definition at line 128 of file mark3c.cpp.

20.5.5.27 Free_Memory()

```
void Free_Memory (
            void * pvObject_ )
```

Free_Memory.

Parameters

*pv↵ Object_*	Pointer to previously allocated block of memory

Definition at line 34 of file mark3c.cpp.

20.5.5.28 Free_Message()

```
void Free_Message (
            Message_t handle )
```

Definition at line 82 of file mark3c.cpp.

20.5.5.29 Free_MessagePool()

```
void Free_MessagePool (
            MessagePool_t handle )
```

Definition at line 106 of file mark3c.cpp.

20.5.5.30 Free_MessageQueue()

```
void Free_MessageQueue (
            MessageQueue_t handle )
```

Definition at line 94 of file mark3c.cpp.

20.5.5.31 Free_Mutex()

```
void Free_Mutex (
            Mutex_t handle )
```

Definition at line 58 of file mark3c.cpp.

20.5.5.32 Free_Notify()

```
void Free_Notify (
            Notify_t handle )
```

Definition at line 118 of file mark3c.cpp.

20.5.5.33 Free_Semaphore()

```
void Free_Semaphore (
            Semaphore_t handle )
```

Definition at line 46 of file mark3c.cpp.

20.5.5.34 Free_Thread()

```
void Free_Thread (
            Thread_t handle )
```

Definition at line 161 of file mark3c.cpp.

20.5.5.35 Free_Timer()

```
void Free_Timer (
            Timer_t handle )
```

Definition at line 172 of file mark3c.cpp.

20.5.5.36 Kernel_DebugPrint()

```
void Kernel_DebugPrint (
            const char * szString_ )
```

KernelDebug_DebugPrint.

See also

void DebugPrint(const char* szString_)

Parameters

*sz↩ String_*	String to print to debug interface

Definition at line 282 of file mark3c.cpp.

20.5.5.37 Kernel_GetTicks()

```
uint32_t Kernel_GetTicks (
            void  )
```

Kernel_GetTicks.

See also

Kernel::GetTicks()

Returns

Number of kernel ticks that have elapsed since boot

Definition at line 216 of file mark3c.cpp.

20.5.5.38 Kernel_Init()

```
void Kernel_Init (
            void  )
```

Kernel_Init.

See also

void Kernel::Init()

Definition at line 180 of file mark3c.cpp.

20.5.5.39 Kernel_IsPanic()

```
bool Kernel_IsPanic (
            void  )
```

Kernel_IsPanic.

See also

bool Kernel::IsPanic()

Returns

Whether or not the kernel is in a panic state

Definition at line 204 of file mark3c.cpp.

20.5.5.40 Kernel_IsStarted()

```
bool Kernel_IsStarted (
            void  )
```

Kernel_IsStarted.

See also

bool Kernel::IsStarted()

Returns

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

Definition at line 192 of file mark3c.cpp.

20.5.5.41 Kernel_Panic()

```
void Kernel_Panic (
            uint16_t u16Cause_ )
```

Kernel_Panic.

See also

void Kernel::Panic(uint16_t u16Cause_)

Parameters

*u16← Cause_*	Reason for the kernel panic

Definition at line 210 of file mark3c.cpp.

20.5.5.42 Kernel_SetDebugPrintFunction()

```
void Kernel_SetDebugPrintFunction (
            kernel_debug_print_t pfPrintFunction_ )
```

Kernel_SetDebugPrintFunction.

See also

void Kernel::SetDebugPrintFunction()

Parameters

*pfPrint← Function_*	Function to use to print debug information from the kernel

Definition at line 276 of file mark3c.cpp.

20.5.5.43 Kernel_SetPanic()

```
void Kernel_SetPanic (
            panic_func_t pfPanic_ )
```

Kernel_SetPanic.

See also

void Kernel::SetPanic(PanicFunc_t pfPanic_)

Parameters

pf← Panic← _	Panic function pointer

Definition at line 198 of file mark3c.cpp.

20.5.5.44 Kernel_Start()

```
void Kernel_Start (
            void  )
```

Kernel_Start.

See also

void Kernel::Start()

Definition at line 186 of file mark3c.cpp.

20.5.5.45 Mailbox_GetFreeSlots()

```
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

Definition at line 832 of file mark3c.cpp.

20.5.5.46 Mailbox_Init()

```
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

Definition at line 769 of file mark3c.cpp.

20.5.5.47 Mailbox_IsEmpty()

```
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

Definition at line 846 of file mark3c.cpp.

20.5.5.48 Mailbox_IsFull()

```
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

Definition at line 839 of file mark3c.cpp.

20.5.5.49 Mailbox_Receive()

```
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.

Definition at line 804 of file mark3c.cpp.

20.5.5.50 Mailbox_ReceiveTail()

```
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.

Definition at line 811 of file mark3c.cpp.

20.5.5.51 Mailbox_Send()

```
bool Mailbox_Send (
            Mailbox_t handle,
            void * pvData_ )
```

Mailbox_Send.

See also

bool Mailbox::Send(void *pvData_)

Parameters

handle	Handle of the mailbox object
*pv←↩ Data_*	Pointer to the data object to send to the mailbox.

Returns

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

Definition at line 776 of file mark3c.cpp.

20.5.5.52 Mailbox_SendTail()

```
bool Mailbox_SendTail (
            Mailbox_t handle,
            void * pvData_ )
```

Mailbox_SendTail.

See also

bool Mailbox::SendTail(void *pvData_)

Parameters

handle	Handle of the mailbox object
*pv←↩ Data_*	Pointer to the data object to send to the mailbox.

Returns

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

Definition at line 783 of file mark3c.cpp.

20.5.5.53 Mailbox_TimedReceive()

```
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.
*u32TimeoutMS_*	Maximum time to wait for delivery.

Returns

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

Definition at line 818 of file mark3c.cpp.

20.5.5.54 Mailbox_TimedReceiveTail()

```
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.

Definition at line 825 of file mark3c.cpp.

20.5.5.55 Mailbox_TimedSend()

```
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.

Definition at line 790 of file mark3c.cpp.

20.5.5.56 Mailbox_TimedSendTail()

```
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.

Definition at line 797 of file mark3c.cpp.

20.5.5.57 Message_GetCode()

```
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

Definition at line 704 of file mark3c.cpp.

20.5.5.58 Message_GetData()

```
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

Definition at line 690 of file mark3c.cpp.

20.5.5.59 Message_Init()

```
void Message_Init (
            Message_t handle )
```

Message_Init.

See also

void Message::Init()

Parameters

handle	Handle of the message object

Definition at line 676 of file mark3c.cpp.

20.5.5.60 Message_SetCode()

```
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

Definition at line 697 of file mark3c.cpp.

20.5.5.61 Message_SetData()

```
void Message_SetData (
            Message_t handle,
            void * pvData_ )
```

Message_SetData.

See also

void Message::SetData(void *pvData_)

Parameters

handle	Handle of the message object
*pv←Data_*	Pointer to the data object to send in the message

Definition at line 683 of file mark3c.cpp.

20.5.5.62 MessagePool_Init()

```
void MessagePool_Init (
            MessagePool_t handle )
```

MessagePool_Init.

See also

void MessagePool::Init()

Parameters

handle	Handle of the message pool object

Definition at line 725 of file mark3c.cpp.

20.5.5.63 MessagePool_Pop()

```
Message_t MessagePool_Pop (
            MessagePool_t handle )
```

MessagePool_Pop.

See also

Message∗ MessagePool::Pop()

Parameters

handle	Handle of the message pool object

Returns

Handle to a Message object, or nullptr on allocation error

Definition at line 739 of file mark3c.cpp.

20.5.5.64 MessagePool_Push()

```
void MessagePool_Push (
            MessagePool_t handle,
            Message_t msg )
```

MessagePool_Push.

See also

void MessagePool::Push(Message∗ pclMessage_)

Parameters

handle	Handle of the message pool object
msg	Message object to return back to the pool

Definition at line 732 of file mark3c.cpp.

20.5.5.65 MessageQueue_GetCount()

```
uint16_t MessageQueue_GetCount (
            MessageQueue_t handle )
```

MessageQueue_GetCount.

See also

uint16_t MessageQueue::GetCount()

Returns

Count of pending messages in the queue.

Definition at line 760 of file mark3c.cpp.

20.5.5.66 MessageQueue_Init()

```
void MessageQueue_Init (
            MessageQueue_t handle )
```

MessageQueue_Init.

See also

void MessageQueue::Init()

Parameters

handle	Handle to the message queue to initialize

Definition at line 711 of file mark3c.cpp.

20.5.5.67 MessageQueue_Receive()

```
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

Definition at line 718 of file mark3c.cpp.

20.5.5.68 MessageQueue_Send()

```
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
h← Message← _	Handle to the message to send to the given queue

Definition at line 753 of file mark3c.cpp.

20.5.5.69 MessageQueue_TimedReceive()

```
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 nullptr on timeout.

Definition at line 746 of file mark3c.cpp.

20.5.5.70 Mutex_Claim()

```
void Mutex_Claim (
            Mutex_t handle )
```

Mutex_Claim.

See also

void Mutex::Claim()

Parameters

handle	Handle of the mutex

Definition at line 515 of file mark3c.cpp.

20.5.5.71 Mutex_Init()

```
void Mutex_Init (
            Mutex_t handle )
```

Mutex_Init.

See also

void Mutex::Init()

Parameters

handle	Handle of the mutex

Definition at line 508 of file mark3c.cpp.

20.5.5.72 Mutex_Release()

```
void Mutex_Release (
            Mutex_t handle )
```

Mutex_Release.

See also

void Mutex::Release()

Parameters

handle	Handle of the mutex

Definition at line 522 of file mark3c.cpp.

20.5.5.73 Mutex_TimedClaim()

```
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

Definition at line 529 of file mark3c.cpp.

20.5.5.74 Notify_Init()

```
void Notify_Init (
            Notify_t handle )
```

Notify_Init.

See also

void Notify::Init()

Parameters

handle	Handle of the notification object

Definition at line 584 of file mark3c.cpp.

20.5.5.75 Notify_Signal()

```
void Notify_Signal (
            Notify_t handle )
```

Notify_Signal.

See also

void Notify::Signal()

Parameters

handle	Handle of the notification object

Definition at line 591 of file mark3c.cpp.

20.5.5.76 Notify_TimedWait()

```
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
*u32WaitTimeMS_*	Maximum time to wait for notification in ms
*pbFlag_*	Flag to set to true on notification

Returns

true on unblock, false on timeout

Definition at line 605 of file mark3c.cpp.

20.5.5.77 Notify_Wait()

```
void Notify_Wait (
            Notify_t handle,
            bool * pbFlag_ )
```

Notify_Wait.

See also

void Notify::Wait(bool *pbFlag_)

Parameters

handle	Handle of the notification object
*pb←↩ Flag_*	Flag to set to true on notification

Definition at line 598 of file mark3c.cpp.

20.5.5.78 ReaderWriterLock_AcquireReader()

```
void ReaderWriterLock_AcquireReader (
            ReaderWriterLock_t handle )
```

ReaderWriterLock_AcquireReader.

See also

void ReaderWriterLock::AcquireReader()

Parameters

handle	Handle of the reader-writer object

Definition at line 897 of file mark3c.cpp.

20.5.5.79 ReaderWriterLock_AcquireWriter()

```
void ReaderWriterLock_AcquireWriter (
            ReaderWriterLock_t handle )
```

ReaderWriterLock_AcquireWriter.

See also

void ReaderWriterLock::AcquireWriter()

Parameters

handle	Handle of the reader-writer object

Definition at line 911 of file mark3c.cpp.

20.5.5.80 ReaderWriterLock_Init()

```
void ReaderWriterLock_Init (
            ReaderWriterLock_t handle )
```

ReaderWriterLock_Init.

See also

void ReaderWriterLock::Init()

Parameters

handle	Handle of the reader-writer object

Definition at line 890 of file mark3c.cpp.

20.5.5.81 ReaderWriterLock_ReleaseReader()

```
void ReaderWriterLock_ReleaseReader (
            ReaderWriterLock_t handle )
```

ReaderWriterLock_ReleaseReader.

See also

void ReaderWriterLock::ReleaseReader()

Parameters

handle	Handle of the reader-writer object

Definition at line 904 of file mark3c.cpp.

20.5.5.82 ReaderWriterLock_ReleaseWriter()

```
void ReaderWriterLock_ReleaseWriter (
            ReaderWriterLock_t handle )
```

ReaderWriterLock_ReleaseWriter.

See also

void ReaderWriterLock::ReleaseWriter()

Parameters

handle	Handle of the reader-writer object

Definition at line 918 of file mark3c.cpp.

20.5.5.83 ReaderWriterLock_TimedAcquireReader()

```
bool ReaderWriterLock_TimedAcquireReader (
            ReaderWriterLock_t handle,
            uint32_t u32TimeoutMs_ )
```

ReaderWriterLock_TimedAcquireReader.

See also

bool ReaderWriterLock::AcquireReader(uint32_t u32TimeoutMs_)

Parameters

handle	Handle of the reader-writer object
*u32TimeoutMs_*	Maximum time to wait for the reader lock before bailing

Returns

true on success, false on timeout

Definition at line 932 of file mark3c.cpp.

20.5.5.84 ReaderWriterLock_TimedAcquireWriter()

```
bool ReaderWriterLock_TimedAcquireWriter (
            ReaderWriterLock_t handle,
            uint32_t u32TimeoutMs_ )
```

ReaderWriterLock_TimedAcquireWriter.

See also

bool ReaderWriterLock::AcquireWriter(uint32_t u32TimeoutMs_)

Parameters

handle	Handle of the reader-writer object
*u32Timeout↵ Ms_*	Maximum time to wait for the writer lock before bailing

Returns

true on success, false on timeout

Definition at line 925 of file mark3c.cpp.

20.5.5.85 Scheduler_Enable()

```
void Scheduler_Enable (
            bool bEnable_ )
```

Scheduler_Enable.

See also

void Scheduler::SetScheduler(bool bEnable_)

Parameters

b↵ Enable↵ _	true to enable, false to disable the scheduler

Definition at line 289 of file mark3c.cpp.

20.5.5.86 Scheduler_GetCurrentThread()

```
Thread_t Scheduler_GetCurrentThread (
            void )
```

Scheduler_GetCurrentThread.

See also

Thread∗ Scheduler::GetCurrentThread()

Returns

Handle of the currently-running thread

Definition at line 301 of file mark3c.cpp.

20.5.5.87 Scheduler_IsEnabled()

```
bool Scheduler_IsEnabled (
            void )
```

Scheduler_IsEnabled.

See also

bool Scheduler::IsEnabled()

Returns

true - scheduler enabled, false - disabled

Definition at line 295 of file mark3c.cpp.

20.5.5.88 Semaphore_Init()

```
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

Definition at line 478 of file mark3c.cpp.

20.5.5.89 Semaphore_Pend()

```
void Semaphore_Pend (
            Semaphore_t handle )
```

Semaphore_Pend.

See also

void Semaphore::Pend()

Parameters

handle	Handle of the semaphore

Definition at line 492 of file mark3c.cpp.

20.5.5.90 Semaphore_Post()

```
void Semaphore_Post (
            Semaphore_t handle )
```

Semaphore_Post.

See also

void Semaphore::Post()

Parameters

handle	Handle of the semaphore

Definition at line 485 of file mark3c.cpp.

20.5.5.91 Semaphore_TimedPend()

```
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

Definition at line 499 of file mark3c.cpp.

20.5.5.92 Thread_CoopYield()

```
void Thread_CoopYield (
            void   )
```

Thread_CoopYield.

See also

void Thread::CoopYield()

Definition at line 415 of file mark3c.cpp.

20.5.5.93 Thread_Exit()

```
void Thread_Exit (
            Thread_t handle )
```

Thread_Exit.

See also

void Thread::Exit()

Parameters

handle	Handle of the thread

Definition at line 381 of file mark3c.cpp.

20.5.5.94 Thread_GetCurPriority()

```
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

Definition at line 354 of file mark3c.cpp.

20.5.5.95 Thread_GetID()

```
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

Definition at line 426 of file mark3c.cpp.

20.5.5.96 Thread_GetPriority()

```
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

Definition at line 348 of file mark3c.cpp.

20.5.5.97 Thread_GetState()

```
thread_state_t Thread_GetState (
             Thread_t handle )
```

Thread_GetState.

See also

ThreadState Thread::GetState()

Parameters

handle	Handle of the thread

Returns

The thread's current execution state

Definition at line 438 of file mark3c.cpp.

20.5.5.98 Thread_Init()

```
void Thread_Init (
             Thread_t handle,
             K_WORD * pwStack_,
             uint16_t u16StackSize_,
             PORT_PRIO_TYPE uXPriority_,
             thread_entry_func_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
*u16Stack←Size_*	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.

Definition at line 310 of file mark3c.cpp.

20.5.5.99 Thread_SetID()

```
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
*u8I←D_*	ID To assign to the thread

Definition at line 420 of file mark3c.cpp.

20.5.5.100 Thread_SetPriority()

```
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
*uX↩ Priority_*	New priority level

Definition at line 374 of file mark3c.cpp.

20.5.5.101 Thread_Sleep()

```
void Thread_Sleep (
            uint32_t u32TimeMs_ )
```

Thread_Sleep.

See also

void Thread::Sleep(uint32_t u32TimeMs_)

Parameters

*u32Time↩ Ms_*	Time in ms to block the thread for

Definition at line 388 of file mark3c.cpp.

20.5.5.102 Thread_Start()

```
void Thread_Start (
            Thread_t handle )
```

Thread_Start.

See also

void Thread::Start()

Parameters

handle	Handle of the thread to start

Definition at line 322 of file mark3c.cpp.

20.5.5.103 Thread_Stop()

```
void Thread_Stop (
            Thread_t handle )
```

Thread_Stop.

See also

void Thread::Stop()

Parameters

handle	Handle of the thread to stop

Definition at line 329 of file mark3c.cpp.

20.5.5.104 Thread_Yield()

```
void Thread_Yield (
            void  )
```

Thread_Yield.

See also

void Thread::Yield()

Definition at line 410 of file mark3c.cpp.

20.5.5.105 Timer_Init()

```
void Timer_Init (
            Timer_t handle )
```

Timer_Init.

See also

void Timer::Init()

Parameters

handle	Handle of the timer

Definition at line 448 of file mark3c.cpp.

20.5.5.106 Timer_Restart()

```
void Timer_Restart (
            Timer_t handle )
```

Timer_Restart.

See also

void Timer::Start()

Parameters

handle	Handle of the timer to restart.

Definition at line 469 of file mark3c.cpp.

20.5.5.107 Timer_Start()

```
void Timer_Start (
            Timer_t handle,
            bool bRepeat_,
            uint32_t u32IntervalMs_,
            timer_callback_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
*u32Interval←Ms_*	Time in ms to expiry
*pfCallback_*	Callback to run on timer expiry
*pvData_*	Data to pass to the callback on expiry

Definition at line 455 of file mark3c.cpp.

20.5.5.108 Timer_Stop()

```
void Timer_Stop (
            Timer_t handle )
```

Timer_Stop.

See also

void Timer::Stop()

Parameters

handle	Handle of the timer

Definition at line 462 of file mark3c.cpp.

20.6 mark3c.h

```
/*===========================================================================
     _____     _____     _____     _____
 ___|    _|__ __|_    |__ __|__   |__ __| __  |__ ______
|    \  /  | ||    \    ||    |     ||  |/ /    ||___   |
|     \/   | ||     \   ||     \    ||  |  \    ||___   |
|__/\__/|__|_||__|\__\ __||__|\__\ __||__|\__\ __||______|
    |_____|      |_____|     |_____|     |_____|

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

Copyright (c) 2012 - 2019 m0slevin, all rights reserved.
See license.txt for more information
===========================================================================*/
#pragma once

#include "mark3cfg.h"
#include "fake_types.h"

#include <stdint.h>
#include <stdbool.h>

#if defined(__cplusplus)
extern "C" {
#endif

//---------------------------------------------------------------------------
// Define a series of handle types to be used in place of the underlying classes
// of Mark3.
#if KERNEL_EVENT_FLAGS
typedef void* EventFlag_t;
#endif                             // #if KERNEL_EVENT_FLAGS
typedef void* Mailbox_t;
typedef void* Message_t;
typedef void* MessagePool_t;
typedef void* MessageQueue_t;
typedef void* Mutex_t;
typedef void* Notify_t;
typedef void* Semaphore_t;
typedef void* Thread_t;
typedef void* Timer_t;
typedef void* ConditionVariable_t;
typedef void* ReaderWriterLock_t;

//---------------------------------------------------------------------------
// Function pointer types used by Kernel APIs
#if KERNEL_THREAD_CREATE_CALLOUT
```

```
typedef void (*thread_create_callout_t)(Thread_t hThread_);
#endif // #if KERNEL_THREAD_CREATE_CALLOUT
#if KERNEL_THREAD_EXIT_CALLOUT
typedef void (*thread_exit_callout_t)(Thread_t hThread_);
#endif // #if KERNEL_THREAD_EXIT_CALLOUT
#if KERNEL_CONTEXT_SWITCH_CALLOUT
typedef void (*thread_context_callout_t)(Thread_t hThread_);
#endif // #if KERNEL_CONTEXT_SWITCH_CALLOUT
typedef void (*kernel_debug_print_t)(const char* szString_);

//---------------------------------------------------------------------------
// Use the sizes of the structs in fake_types.h to generate opaque object-blobs
// that get instantiated as kernel objects (from the C++ code) later.
#define THREAD_SIZE (sizeof(Fake_Thread))
#define TIMER_SIZE (sizeof(Fake_Timer))
#define SEMAPHORE_SIZE (sizeof(Fake_Semaphore))
#define MUTEX_SIZE (sizeof(Fake_Mutex))
#define MESSAGE_SIZE (sizeof(Fake_Message))
#define MESSAGEQUEUE_SIZE (sizeof(Fake_MessageQueue))
#define MAILBOX_SIZE (sizeof(Fake_Mailbox))
#define NOTIFY_SIZE (sizeof(Fake_Notify))
#if KERNEL_EVENT_FLAGS
#define EVENTFLAG_SIZE (sizeof(Fake_EventFlag))
#endif // #if KERNEL_EVENT_FLAGS
#define MESSAGEPOOL_SIZE (sizeof(Fake_MessagePool))
#define CONDITIONVARIABLE_SIZE (sizeof(Fake_ConditionVariable))
#define READERWRITERLOCK_SIZE (sizeof(Fake_ReaderWriterLock))

#if KERNEL_EVENT_FLAGS
//---------------------------------------------------------------------------
typedef enum {
    EVENT_FLAG_ALL_SET,
    EVENT_FLAG_ANY_SET,
    EVENT_FLAG_ALL_CLEAR,
    EVENT_FlAG_ANY_CLEAR,
    EVENT_FLAG_PENDING_UNBLOCK
} event_flag_operation_t;
#endif // #if KERNEL_EVENT_FLAGS

//---------------------------------------------------------------------------
typedef enum {
    THREAD_STATE_EXIT = 0,
    THREAD_STATE_READY,
    THREAD_STATE_BLOCKED,
    THREAD_STATE_STOP,
    THREAD_STATE_INVALID
} thread_state_t;

//---------------------------------------------------------------------------
// Macros for declaring opaque buffers of an appropriate size for the given
// kernel objects
#define TOKEN_1(x, y) x##y
#define TOKEN_2(x, y) TOKEN_1(x, y)

// Ensure that opaque buffers are sized to the nearest word - which is
// a platform-dependent value.
#define WORD_ROUND(x) (((x) + (sizeof(K_WORD) - 1)) / sizeof(K_WORD))

#define DECLARE_THREAD(name) \
    K_WORD  TOKEN_2(__thread_, name)[WORD_ROUND(THREAD_SIZE)]; \
    Thread_t name = (Thread_t)TOKEN_2(__thread_, name);

#define DECLARE_TIMER(name) \
    K_WORD TOKEN_2(__timer_, name)[WORD_ROUND(TIMER_SIZE)]; \
    Timer_t name = (Timer_t)TOKEN_2(__timer_, name);

#define DECLARE_SEMAPHORE(name) \
    K_WORD     TOKEN_2(__semaphore_, name)[WORD_ROUND(SEMAPHORE_SIZE)]; \
    Semaphore_t name = (Semaphore_t)TOKEN_2(__semaphore_, name);

#define DECLARE_MUTEX(name) \
    K_WORD TOKEN_2(__mutex_, name)[WORD_ROUND(MUTEX_SIZE)]; \
    Mutex_t name = (Mutex_t)TOKEN_2(__mutex_, name);

#define DECLARE_MESSAGE(name) \
    K_WORD   TOKEN_2(__message_, name)[WORD_ROUND(MESSAGE_SIZE)]; \
    Message_t name = (Message_t)TOKEN_2(__message_, name);
```

```

#define DECLARE_MESSAGEPOOL(name) \
    K_WORD       TOKEN_2(__messagepool_, name)[WORD_ROUND(MESSAGEPOOL_SIZE)]; \
    MessagePool_t name = (MessagePool_t)TOKEN_2(__messagepool_, name);

#define DECLARE_MESSAGEQUEUE(name) \
    K_WORD        TOKEN_2(__messagequeue_, name)[WORD_ROUND(MESSAGEQUEUE_SIZE)]; \
    MessageQueue_t name = (MessageQueue_t)TOKEN_2(__messagequeue_, name);

#define DECLARE_MAILBOX(name) \
    K_WORD   TOKEN_2(__mailbox_, name)[WORD_ROUND(MAILBOX_SIZE)]; \
    Mailbox_t name = (Mailbox_t)TOKEN_2(__mailbox_, name);

#define DECLARE_NOTIFY(name) \
    K_WORD  TOKEN_2(__notify_, name)[WORD_ROUND(NOTIFY_SIZE)]; \
    Notify_t name = (Notify_t)TOKEN_2(__notify_, name);

#if KERNEL_EVENT_FLAGS
#define DECLARE_EVENTFLAG(name) \
    K_WORD     TOKEN_2(__eventflag_, name)[WORD_ROUND(EVENTFLAG_SIZE)]; \
    EventFlag_t name = (EventFlag_t)TOKEN_2(__eventflag_, name);
#endif // #if KERNEL_EVENT_FLAGS

#define DECLARE_CONDITIONVARIABLE(name) \
    K_WORD             TOKEN_2(__condvar_, name)[WORD_ROUND(EVENTFLAG_SIZE)]; \
    ConditionVariable_t name = (ConditionVariable_t)TOKEN_2(__condvar_, name);

#define DECLARE_READERWRITERLOCK(name) \
    K_WORD            TOKEN_2(__readerwriterlock_, name)[WORD_ROUND(EVENTFLAG_SIZE)]; \
    ReaderWriterLock_t name = (ReaderWriterLock_t)TOKEN_2(__readerwriterlock_, name);

//---------------------------------------------------------------------------
// Allocate-once Memory managment APIs
void* Alloc_Memory(size_t eSize_);

void Free_Memory(void* pvObject_);

Semaphore_t Alloc_Semaphore(void);
void        Free_Semaphore(Semaphore_t handle);

Mutex_t Alloc_Mutex(void);
void    Free_Mutex(Mutex_t handle);

#if KERNEL_EVENT_FLAGS

EventFlag_t Alloc_EventFlag(void);
void        Free_EventFlag(EventFlag_t handle);
#endif // #if KERNEL_EVENT_FLAGS

Message_t Alloc_Message(void);
void      Free_Message(Message_t handle);

MessageQueue_t Alloc_MessageQueue(void);
void           Free_MessageQueue(MessageQueue_t handle);

MessagePool_t Alloc_MessagePool(void);
void          Free_MessagePool(MessagePool_t handle);

Notify_t Alloc_Notify(void);
void     Free_Notify(Notify_t handle);

Mailbox_t Alloc_Mailbox(void);
void      Free_Mailbox(Mailbox_t handle);

Thread_t Alloc_Thread(void);
void     Free_Thread(Thread_t handle);

Timer_t Alloc_Timer(void);
void    Free_Timer(Timer_t handle);

//---------------------------------------------------------------------------
// Kernel APIs
void Kernel_Init(void);
```

```
void Kernel_Start(void);
bool Kernel_IsStarted(void);

typedef void (*panic_func_t)(uint16_t u16PanicCode_);
void Kernel_SetPanic(panic_func_t pfPanic_);
bool Kernel_IsPanic(void);
void Kernel_Panic(uint16_t u16Cause_);

#if KERNEL_THREAD_CREATE_CALLOUT

void Kernel_SetThreadCreateCallout(thread_create_callout_t pfCreate_);
#endif // #if KERNEL_THREAD_CREATE_CALLOUT

#if KERNEL_THREAD_EXIT_CALLOUT

void Kernel_SetThreadExitCallout(thread_exit_callout_t pfExit_);
#endif // #if KERNEL_THREAD_EXIT_CALLOUT

#if KERNEL_CONTEXT_SWITCH_CALLOUT

void Kernel_SetThreadContextSwitchCallout(thread_context_callout_t pfContext_);
#endif // #if KERNEL_CONTEXT_SWITCH_CALLOUT

#if KERNEL_THREAD_CREATE_CALLOUT

thread_create_callout_t Kernel_GetThreadCreateCallout(void);
#endif // #if KERNEL_THREAD_CREATE_CALLOUT

#if KERNEL_THREAD_EXIT_CALLOUT

thread_exit_callout_t Kernel_GetThreadExitCallout(void);
#endif // #if KERNEL_THREAD_EXIT_CALLOUT

#if KERNEL_CONTEXT_SWITCH_CALLOUT

thread_context_callout_t Kernel_GetThreadContextSwitchCallout(void);
#endif // #if KERNEL_CONTEXT_SWITCH_CALLOUT

#if KERNEL_STACK_CHECK

void Kernel_SetStackGuardThreshold(uint16_t u16Threshold_);

uint16_t Kernel_GetStackGuardThreshold(void);
#endif // #if KERNEL_STACK_CHECK

uint32_t Kernel_GetTicks(void);

//---------------------------------------------------------------------------
// Scheduler APIs
void Scheduler_Enable(bool bEnable_);
bool Scheduler_IsEnabled(void);
Thread_t Scheduler_GetCurrentThread(void);

typedef void (*thread_entry_func_t)(void* pvArg_);
//---------------------------------------------------------------------------
// Thread APIs
void Thread_Init(Thread_t            handle,
                 K_WORD*             pwStack_,
                 uint16_t            u16StackSize_,
                 PORT_PRIO_TYPE      uXPriority_,
                 thread_entry_func_t pfEntryPoint_,
                 void*               pvArg_);
void Thread_Start(Thread_t handle);
void Thread_Stop(Thread_t handle);

#if KERNEL_NAMED_THREADS

void Thread_SetName(Thread_t handle, const char* szName_);
const char* Thread_GetName(Thread_t handle);
#endif // #if KERNEL_NAMED_THREADS

PORT_PRIO_TYPE Thread_GetPriority(Thread_t handle);
PORT_PRIO_TYPE Thread_GetCurPriority(Thread_t handle);

#if KERNEL_ROUND_ROBIN

void Thread_SetQuantum(Thread_t handle, uint16_t u16Quantum_);
uint16_t Thread_GetQuantum(Thread_t handle);
#endif // #if KERNEL_ROUND_ROBIN

void Thread_SetPriority(Thread_t handle, PORT_PRIO_TYPE uXPriority_);
void Thread_Exit(Thread_t handle);
void Thread_Sleep(uint32_t u32TimeMs_);

#if KERNEL_EXTENDED_CONTEXT

void* Thread_GetExtendedContext(Thread_t handle);
```

```
void Thread_SetExtendedContext(Thread_t handle, void* pvData_);
#endif // #if KERNEL_EXTENDED_CONTEXT

void Thread_Yield(void);

void Thread_CoopYield(void);

void Thread_SetID(Thread_t handle, uint8_t u8ID_);
uint8_t Thread_GetID(Thread_t handle);

#if KERNEL_STACK_CHECK

uint16_t Thread_GetStackSlack(Thread_t handle);
#endif // #if KERNEL_STACK_CHECK

thread_state_t Thread_GetState(Thread_t handle);

//---------------------------------------------------------------------------
// Timer APIs
typedef void (*timer_callback_t)(Thread_t hOwner_, void* pvData_);
void Timer_Init(Timer_t handle);
void Timer_Start(Timer_t handle, bool bRepeat_, uint32_t u32IntervalMs_,
 timer_callback_t pfCallback_, void* pvData_);

void Timer_Restart(Timer_t handle);

void Timer_Stop(Timer_t handle);

//---------------------------------------------------------------------------
// Semaphore APIs
void Semaphore_Init(Semaphore_t handle, uint16_t u16InitVal_, uint16_t u16MaxVal_);
void Semaphore_Post(Semaphore_t handle);
void Semaphore_Pend(Semaphore_t handle);
bool Semaphore_TimedPend(Semaphore_t handle, uint32_t u32WaitTimeMS_);

//---------------------------------------------------------------------------
// Mutex APIs
void Mutex_Init(Mutex_t handle);
void Mutex_Claim(Mutex_t handle);
void Mutex_Release(Mutex_t handle);
bool Mutex_TimedClaim(Mutex_t handle, uint32_t u32WaitTimeMS_);

#if KERNEL_EVENT_FLAGS
//---------------------------------------------------------------------------
// EventFlag APIs
void EventFlag_Init(EventFlag_t handle);
uint16_t EventFlag_Wait(EventFlag_t handle, uint16_t u16Mask_, event_flag_operation_t eMode_);
uint16_t EventFlag_TimedWait(EventFlag_t handle, uint16_t u16Mask_, event_flag_operation_t eMode_, uint32_t
 u32TimeMS_);
void EventFlag_Set(EventFlag_t handle, uint16_t u16Mask_);
void EventFlag_Clear(EventFlag_t handle, uint16_t u16Mask_);
uint16_t EventFlag_GetMask(EventFlag_t handle);
#endif // #if KERNEL_EVENT_FLAGS

//---------------------------------------------------------------------------
// Notification APIs
void Notify_Init(Notify_t handle);
void Notify_Signal(Notify_t handle);
void Notify_Wait(Notify_t handle, bool* pbFlag_);
bool Notify_TimedWait(Notify_t handle, uint32_t u32WaitTimeMS_, bool* pbFlag_);

//---------------------------------------------------------------------------
// Atomic Functions
uint8_t Atomic_Set8(uint8_t* pu8Source_, uint8_t u8Val_);
uint16_t Atomic_Set16(uint16_t* pu16Source_, uint16_t u16Val_);
uint32_t Atomic_Set32(uint32_t* pu32Source_, uint32_t u32Val_);
uint8_t Atomic_Add8(uint8_t* pu8Source_, uint8_t u8Val_);
uint16_t Atomic_Add16(uint16_t* pu16Source_, uint16_t u16Val_);
uint32_t Atomic_Add32(uint32_t* pu32Source_, uint32_t u32Val_);
uint8_t Atomic_Sub8(uint8_t* pu8Source_, uint8_t u8Val_);
uint16_t Atomic_Sub16(uint16_t* pu16Source_, uint16_t u16Val_);
uint32_t Atomic_Sub32(uint32_t* pu32Source_, uint32_t u32Val_);
bool Atomic_TestAndSet(bool* pbLock);

//---------------------------------------------------------------------------
// Message/Message Queue APIs
void Message_Init(Message_t handle);
void Message_SetData(Message_t handle, void* pvData_);
void* Message_GetData(Message_t handle);
void Message_SetCode(Message_t handle, uint16_t u16Code_);
uint16_t Message_GetCode(Message_t handle);
void MessageQueue_Init(MessageQueue_t handle);
Message_t MessageQueue_Receive(MessageQueue_t handle);
Message_t MessageQueue_TimedReceive(MessageQueue_t handle, uint32_t u32TimeWaitMS_
);

```

```
void MessageQueue_Send(MessageQueue_t handle, Message_t hMessage_);

uint16_t MessageQueue_GetCount(MessageQueue_t handle);

void MessagePool_Init(MessagePool_t handle);

void MessagePool_Push(MessagePool_t handle, Message_t msg);

Message_t MessagePool_Pop(MessagePool_t handle);

//---------------------------------------------------------------------------
// Mailbox APIs

void Mailbox_Init(Mailbox_t handle, void* pvBuffer_, uint16_t u16BufferSize_, uint16_t u16ElementSize_);

bool Mailbox_Send(Mailbox_t handle, void* pvData_);

bool Mailbox_SendTail(Mailbox_t handle, void* pvData_);

bool Mailbox_TimedSend(Mailbox_t handle, void* pvData_, uint32_t u32TimeoutMS_);

bool Mailbox_TimedSendTail(Mailbox_t handle, void* pvData_, uint32_t u32TimeoutMS_);

void Mailbox_Receive(Mailbox_t handle, void* pvData_);

void Mailbox_ReceiveTail(Mailbox_t handle, void* pvData_);

bool Mailbox_TimedReceive(Mailbox_t handle, void* pvData_, uint32_t u32TimeoutMS_);

bool Mailbox_TimedReceiveTail(Mailbox_t handle, void* pvData_, uint32_t u32TimeoutMS_);

uint16_t Mailbox_GetFreeSlots(Mailbox_t handle);

bool Mailbox_IsFull(Mailbox_t handle);

bool Mailbox_IsEmpty(Mailbox_t handle);

//---------------------------------------------------------------------------
// Condition Variables
void ConditionVariable_Init(ConditionVariable_t handle);

void ConditionVariable_Wait(ConditionVariable_t handle, Mutex_t hMutex_);

void ConditionVariable_Signal(ConditionVariable_t handle);

void ConditionVariable_Broadcast(ConditionVariable_t handle);

bool ConditionVariable_TimedWait(ConditionVariable_t handle, Mutex_t hMutex_, uint32_t u32WaitTimeMS_);

//---------------------------------------------------------------------------
// Reader-writer locks
void ReaderWriterLock_Init(ReaderWriterLock_t handle);

void ReaderWriterLock_AcquireReader(ReaderWriterLock_t handle);

void ReaderWriterLock_ReleaseReader(ReaderWriterLock_t handle);

void ReaderWriterLock_AcquireWriter(ReaderWriterLock_t handle);

void ReaderWriterLock_ReleaseWriter(ReaderWriterLock_t handle);

bool ReaderWriterLock_TimedAcquireWriter(ReaderWriterLock_t handle, uint32_t u32TimeoutMs_);

bool ReaderWriterLock_TimedAcquireReader(ReaderWriterLock_t handle, uint32_t u32TimeoutMs_);

void Kernel_SetDebugPrintFunction(kernel_debug_print_t pfPrintFunction_);

void Kernel_DebugPrint(const char* szString_);

#if defined(__cplusplus)
}
#endif
```

20.7 /home/moslevin/projects/m3-repo/kernel/lib/memutil/memutil.cpp File Reference

Implementation of memory, string, and conversion routines.

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

Namespaces

- Mark3

20.7.1 Detailed Description

Implementation of memory, string, and conversion routines.

Definition in file memutil.cpp.

20.8 memutil.cpp

```
/*===========================================================================
     _____        _____        _____        _____
 ___|    _|__  __|_    |__  __|__   |__  __| __  |__  ______
|    \  /  | ||    \      ||     |     ||  |/ /     ||___   |
|     \/   | ||     \     ||     \     ||     \     ||___   |
|__/\__/|__|_||__|\__\  __||__|\__\  __||__|\__\  __||______|
    |_____|      |_____|      |_____|      |_____|

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

Copyright (c) 2012 - 2019 m0slevin, all rights reserved.
See license.txt for more information
===========================================================================*/
#include "kerneltypes.h"
#include "mark3cfg.h"
#include "memutil.h"
#include "kerneldebug.h"

namespace Mark3
{
//---------------------------------------------------------------------------
void MemUtil::DecimalToHex(uint8_t u8Data_, char* szText_)
{
    uint8_t u8Tmp = u8Data_;
    uint8_t u8Max;

    KERNEL_ASSERT(szText_);

    if (u8Tmp >= 0x10) {
        u8Max = 2;
    } else {
        u8Max = 1;
    }

    u8Tmp          = u8Data_;
    szText_[u8Max] = 0;
    while ((u8Max--) != 0u) {
        if ((u8Tmp & 0x0F) <= 9) {
            szText_[u8Max] = '0' + (u8Tmp & 0x0F);
        } else {
            szText_[u8Max] = 'A' + ((u8Tmp & 0x0F) - 10);
        }
        u8Tmp >>= 4;
    }
}

//---------------------------------------------------------------------------
void MemUtil::DecimalToHex(uint16_t u16Data_, char* szText_)
{
    uint16_t u16Tmp     = u16Data_;
    uint16_t u16Max     = 1;
    uint16_t u16Compare = 0x0010;
```

```
    KERNEL_ASSERT(szText_);

    while (u16Data_ > u16Compare && u16Max < 4) {
        u16Max++;
        u16Compare <<= 4;
    }

    u16Tmp          = u16Data_;
    szText_[u16Max] = 0;
    while ((u16Max--) != 0u) {
        if ((u16Tmp & 0x0F) <= 9) {
            szText_[u16Max] = '0' + (u16Tmp & 0x0F);
        } else {
            szText_[u16Max] = 'A' + ((u16Tmp & 0x0F) - 10);
        }
        u16Tmp >>= 4;
    }
}

//---------------------------------------------------------------------------
void MemUtil::DecimalToHex(uint32_t u32Data_, char* szText_)
{
    uint32_t u32Tmp     = u32Data_;
    uint32_t u32Max     = 1;
    uint32_t u32Compare = 0x0010;

    KERNEL_ASSERT(szText_);

    while (u32Data_ > u32Compare && u32Max < 8) {
        u32Max++;
        u32Compare <<= 4;
    }

    u32Tmp          = u32Data_;
    szText_[u32Max] = 0;
    while ((u32Max--) != 0u) {
        if ((u32Tmp & 0x0F) <= 9) {
            szText_[u32Max] = '0' + (u32Tmp & 0x0F);
        } else {
            szText_[u32Max] = 'A' + ((u32Tmp & 0x0F) - 10);
        }
        u32Tmp >>= 4;
    }
}

//---------------------------------------------------------------------------
void MemUtil::DecimalToHex(uint64_t u64Data_, char* szText_)
{
    uint64_t u64Tmp     = u64Data_;
    uint64_t u64Max     = 1;
    uint64_t u64Compare = 0x0010;

    KERNEL_ASSERT(szText_);

    while (u64Data_ > u64Compare && u64Max < 8) {
        u64Max++;
        u64Compare <<= 4;
    }

    u64Tmp          = u64Data_;
    szText_[u64Max] = 0;
    while ((u64Max--) != 0u) {
        if ((u64Tmp & 0x0F) <= 9) {
            szText_[u64Max] = '0' + (u64Tmp & 0x0F);
        } else {
            szText_[u64Max] = 'A' + ((u64Tmp & 0x0F) - 10);
        }
        u64Tmp >>= 4;
    }
}

//---------------------------------------------------------------------------
void MemUtil::DecimalToString(uint8_t u8Data_, char* szText_)
{
    uint8_t u8Tmp = u8Data_;
    uint8_t u8Max;

    KERNEL_ASSERT(szText_);

    // Find max index to print...
    if (u8Data_ >= 100) {
        u8Max = 3;
    } else if (u8Data_ >= 10) {
        u8Max = 2;
    } else {
        u8Max = 1;
```

```
    }

    szText_[u8Max] = 0;
    while ((u8Max--) != 0u) {
        szText_[u8Max] = '0' + (u8Tmp % 10);
        u8Tmp /= 10;
    }
}

//---------------------------------------------------------------------------
void MemUtil::DecimalToString(uint16_t u16Data_, char* szText_)
{
    uint16_t u16Tmp     = u16Data_;
    uint16_t u16Max     = 1;
    uint16_t u16Compare = 10;

    KERNEL_ASSERT(szText_);

    while (u16Data_ >= u16Compare && u16Max < 5) {
        u16Compare *= 10;
        u16Max++;
    }

    szText_[u16Max] = 0;
    while ((u16Max--) != 0u) {
        szText_[u16Max] = '0' + (u16Tmp % 10);
        u16Tmp /= 10;
    }
}

//---------------------------------------------------------------------------
void MemUtil::DecimalToString(uint32_t u32Data_, char* szText_)
{
    uint32_t u32Tmp     = u32Data_;
    uint32_t u32Max     = 1;
    uint32_t u32Compare = 10;

    KERNEL_ASSERT(szText_);

    while (u32Data_ >= u32Compare && u32Max < 12) {
        u32Compare *= 10;
        u32Max++;
    }

    szText_[u32Max] = 0;
    while ((u32Max--) != 0u) {
        szText_[u32Max] = '0' + (u32Tmp % 10);
        u32Tmp /= 10;
    }
}

//---------------------------------------------------------------------------
void MemUtil::DecimalToString(uint64_t u64Data_, char* szText_)
{
    uint64_t u64Tmp     = u64Data_;
    uint64_t u64Max     = 1;
    uint64_t u64Compare = 10;

    KERNEL_ASSERT(szText_);

    while (u64Data_ >= u64Compare && u64Max < 12) {
        u64Compare *= 10;
        u64Max++;
    }

    szText_[u64Max] = 0;
    while ((u64Max--) != 0u) {
        szText_[u64Max] = '0' + (u64Tmp % 10);
        u64Tmp /= 10;
    }
}

//---------------------------------------------------------------------------
bool MemUtil::StringToDecimal8(const char* szText_, uint8_t* pu8Out_)
{
    uint8_t u8Tmp = 0;
    uint8_t u8Len = 0;

    for (uint8_t i = 0; i < 4; i++) {
        if (szText_[i] == 0) {
            if (i == 0) {
                return false;
            }
            u8Len = i;
            break;
        }
    }
```

```
    for (uint8_t i = 0; i < u8Len; i++) {
        if ((szText_[i] < '0') || (szText_[i] > '9')) {
            return false;
        }
        u8Tmp *= 10;
        u8Tmp += szText_[i] - '0';
    }
    *pu8Out_ = u8Tmp;

    return true;
}

//---------------------------------------------------------------------------
bool MemUtil::StringToDecimal16(const char* szText_, uint16_t* pu16Out_)
{
    uint16_t u16Tmp = 0;
    uint16_t u16Len = 0;

    for (uint8_t i = 0; i < 6; i++) {
        if (szText_[i] == 0) {
            if (i == 0) {
                return false;
            }
            u16Len = i;
            break;
        }
    }

    for (uint8_t i = 0; i < u16Len; i++) {
        if ((szText_[i] < '0') || (szText_[i] > '9')) {
            return false;
        }
        u16Tmp *= 10;
        u16Tmp += szText_[i] - '0';
    }
    *pu16Out_ = u16Tmp;

    return true;
}

//---------------------------------------------------------------------------
bool MemUtil::StringToDecimal32(const char* szText_, uint32_t* pu32Out_)
{
    uint32_t u32Tmp = 0;
    uint32_t u32Len = 0;

    for (uint8_t i = 0; i < 11; i++) {
        if (szText_[i] == 0) {
            if (i == 0) {
                return false;
            }
            u32Len = i;
            break;
        }
    }

    for (uint8_t i = 0; i < u32Len; i++) {
        if ((szText_[i] < '0') || (szText_[i] > '9')) {
            return false;
        }
        u32Tmp *= 10;
        u32Tmp += szText_[i] - '0';
    }
    *pu32Out_ = u32Tmp;

    return true;
}

//---------------------------------------------------------------------------
bool MemUtil::StringToDecimal64(const char* szText_, uint64_t* pu64Out_)
{
    uint64_t u64Tmp = 0;
    uint64_t u64Len = 0;

    for (uint8_t i = 0; i < 21; i++) {
        if (szText_[i] == 0) {
            if (i == 0) {
                return false;
            }
            u64Len = i;
            break;
        }
    }

    for (uint8_t i = 0; i < u64Len; i++) {
        if ((szText_[i] < '0') || (szText_[i] > '9')) {
```

```
            return false;
        }
        u64Tmp *= 10;
        u64Tmp += szText_[i] - '0';
    }
    *pu64Out_ = u64Tmp;

    return true;
}

//---------------------------------------------------------------------------
// Basic checksum routines
uint8_t MemUtil::Checksum8(const void* pvSrc_, uint16_t u16Len_)
{
    uint8_t  u8Ret   = 0;
    uint8_t* pcData = (uint8_t*)pvSrc_;

    KERNEL_ASSERT(pvSrc_);

    // 8-bit CRC, computed byte at a time
    while ((u16Len_--) != 0u) { u8Ret += *pcData++; }
    return u8Ret;
}

//---------------------------------------------------------------------------
uint16_t MemUtil::Checksum16(const void* pvSrc_, uint16_t u16Len_)
{
    uint16_t u16Ret = 0;
    uint8_t* pcData = (uint8_t*)pvSrc_;

    KERNEL_ASSERT(pvSrc_);

    // 16-bit CRC, computed byte at a time
    while ((u16Len_--) != 0u) { u16Ret += *pcData++; }
    return u16Ret;
}

//---------------------------------------------------------------------------
// Basic string routines
uint16_t MemUtil::StringLength(const char* szStr_)
{
    uint8_t* pcData = (uint8_t*)szStr_;
    uint16_t u16Len = 0;

    KERNEL_ASSERT(szStr_);

    while (*pcData++ != 0u) { u16Len++; }
    return u16Len;
}

//---------------------------------------------------------------------------
bool MemUtil::CompareStrings(const char* szStr1_, const char* szStr2_)
{
    char* szTmp1 = (char*)szStr1_;
    char* szTmp2 = (char*)szStr2_;

    KERNEL_ASSERT(szStr1_);
    KERNEL_ASSERT(szStr2_);

    while ((*szTmp1 != 0) && (*szTmp2 != 0)) {
        if (*szTmp1++ != *szTmp2++) {
            return false;
        }
    }

    // Both terminate at the same length
    return ((*szTmp1) == 0) && ((*szTmp2) == 0);
}

//---------------------------------------------------------------------------
bool MemUtil::CompareStrings(const char* szStr1_, const char* szStr2_, uint16_t u16Length_)
{
    char* szTmp1 = (char*)szStr1_;
    char* szTmp2 = (char*)szStr2_;

    while (((*szTmp1 != 0) && (*szTmp2 != 0)) && (u16Length_ != 0u)) {
        if (*szTmp1++ != *szTmp2++) {
            return false;
        }
        u16Length_--;
    }

    // Both terminate at the same length
    return (((*szTmp1) == 0) && ((*szTmp2) == 0)) || (u16Length_ == 0u);
}

```

```
//-----------------------------------------------------------------------------
void MemUtil::CopyMemory(void* pvDst_, const void* pvSrc_, uint16_t u16Len_)
{
    char* szDst = (char*)pvDst_;
    char* szSrc = (char*)pvSrc_;

    KERNEL_ASSERT(pvDst_);
    KERNEL_ASSERT(pvSrc_);

    // Run through the strings verifying that each character matches
    // and the lengths are the same.
    while ((u16Len_--) != 0u) { *szDst++ = *szSrc++; }
}

//-----------------------------------------------------------------------------
void MemUtil::CopyString(char* szDst_, const char* szSrc_)
{
    char* szDst = (char*)szDst_;
    char* szSrc = (char*)szSrc_;

    KERNEL_ASSERT(szDst_);
    KERNEL_ASSERT(szSrc_);

    // Run through the strings verifying that each character matches
    // and the lengths are the same.
    while (*szSrc != 0) { *szDst++ = *szSrc++; }
}

//-----------------------------------------------------------------------------
int16_t MemUtil::StringSearch(const char* szBuffer_, const char* szPattern_)
{
    char*   szTmpPat = (char*)szPattern_;
    int16_t i16Idx   = 0;
    int16_t i16Start;
    KERNEL_ASSERT(szBuffer_);
    KERNEL_ASSERT(szPattern_);

    // Run through the big buffer looking for a match of the pattern
    while (szBuffer_[i16Idx] != 0) {
        // Reload the pattern
        i16Start = i16Idx;
        szTmpPat = (char*)szPattern_;
        while ((*szTmpPat != 0) && (szBuffer_[i16Idx] != 0)) {
            if (*szTmpPat != szBuffer_[i16Idx]) {
                break;
            }
            szTmpPat++;
            i16Idx++;
        }
        // Made it to the end of the pattern, it's a match.
        if (*szTmpPat == '\0') {
            return i16Start;
        }
        i16Idx++;
    }

    return -1;
}

//-----------------------------------------------------------------------------
bool MemUtil::CompareMemory(const void* pvMem1_, const void* pvMem2_, uint16_t
u16Len_)
{
    char* szTmp1 = (char*)pvMem1_;
    char* szTmp2 = (char*)pvMem2_;

    KERNEL_ASSERT(pvMem1_);
    KERNEL_ASSERT(pvMem2_);

    // Run through the strings verifying that each character matches
    // and the lengths are the same.
    while ((u16Len_--) != 0u) {
        if (*szTmp1++ != *szTmp2++) {
            return false;
        }
    }
    return true;
}

//-----------------------------------------------------------------------------
void MemUtil::SetMemory(void* pvDst_, uint8_t u8Val_, uint16_t u16Len_)
{
    char* szDst = (char*)pvDst_;

    KERNEL_ASSERT(pvDst_);

    while ((u16Len_--) != 0u) { *szDst++ = u8Val_; }
```

```
}

//---------------------------------------------------------------------------
uint8_t MemUtil::Tokenize(const char* szBuffer_, Token_t* pastTokens_, uint8_t u8MaxTokens_)
{
    uint8_t u8CurrArg = 0;
    uint8_t u8LastArg = 0;
    uint8_t i         = 0;

    bool bEscape = false;

    KERNEL_ASSERT(szBuffer_);
    KERNEL_ASSERT(pastTokens_);

    while (szBuffer_[i] != 0) {
        //-- Handle unescaped quotes
        if (szBuffer_[i] == '\"') {
            bEscape = !bEscape;
            i++;
            continue;
        }

        //-- Handle all escaped chars - by ignoring them
        if (szBuffer_[i] == '\\') {
            i++;
            if (szBuffer_[i] != 0) {
                i++;
            }
            continue;
        }

        //-- Process chars based on current escape characters
        if (bEscape) {
            // Everything within the quote is treated as literal, but escaped chars are still treated the same
            i++;
            continue;
        }

        //-- Non-escaped case
        if (szBuffer_[i] != ' ') {
            i++;
            continue;
        }

        pastTokens_[u8CurrArg].pcToken = &(szBuffer_[u8LastArg]);
        pastTokens_[u8CurrArg].u8Len   = i - u8LastArg;
        u8CurrArg++;
        if (u8CurrArg >= u8MaxTokens_) {
            return u8MaxTokens_;
        }

        i++;
        while (szBuffer_[i] == ' ') { i++; }

        u8LastArg = i;
    }
    if ((i != 0u) && (szBuffer_[i] == 0) && ((i - u8LastArg) != 0)) {
        pastTokens_[u8CurrArg].pcToken = &(szBuffer_[u8LastArg]);
        pastTokens_[u8CurrArg].u8Len   = i - u8LastArg;
        u8CurrArg++;
    }
    return u8CurrArg;
}
} // namespace Mark3
```

20.9 /home/moslevin/projects/m3-repo/kernel/lib/memutil/public/memutil.h File Reference

Utility class containing memory, string, and conversion routines.

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

Classes

- struct Mark3::Token_t

  Token descriptor struct format.
- class Mark3::MemUtil

  String and Memory manipu32ation class.

Namespaces

- Mark3

20.9.1 Detailed Description

Utility class containing memory, string, and conversion routines.

Definition in file memutil.h.

20.10 memutil.h

```
/*===========================================================================
     _____        _____        _____        _____
 ___|    _|__  __|_    |__  __|__   |__  __| __  |__  ______
|    \  /  | ||    \      ||     |     ||  |/ /     ||___   |
|     \/   | ||     \     ||     \     ||     \     ||___   |
|__/\__/|__|_||__|\__\  __||__|\__\  __||__|\__\  __||______|
    |_____|      |_____|      |_____|      |_____|

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

Copyright (c) 2012 - 2019 m0slevin, all rights reserved.
See license.txt for more information
=========================================================================== */
#pragma once

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

namespace Mark3
{
//---------------------------------------------------------------------------
typedef struct {
    const char* pcToken;
    uint8_t     u8Len;
} Token_t;

//---------------------------------------------------------------------------
class MemUtil
{
public:
    //-----------------------------------------------------------------------
    static void DecimalToHex(uint8_t u8Data_, char* szText_);
    static void DecimalToHex(uint16_t u16Data_, char* szText_);
    static void DecimalToHex(uint32_t u32Data_, char* szText_);
    static void DecimalToHex(uint64_t u64Data_, char* szText_);

    //-----------------------------------------------------------------------
    static void DecimalToString(uint8_t u8Data_, char* szText_);
    static void DecimalToString(uint16_t u16Data_, char* szText_);
    static void DecimalToString(uint32_t u32Data_, char* szText_);
    static void DecimalToString(uint64_t u64Data_, char* szText_);

    //-----------------------------------------------------------------------
    static bool StringToDecimal8(const char* szText_, uint8_t* pu8Out_);
    static bool StringToDecimal16(const char* szText_, uint16_t* pu16Out_);
    static bool StringToDecimal32(const char* szText_, uint32_t* pu32Out_);
    static bool StringToDecimal64(const char* szText_, uint64_t* pu64Out_);

    //-----------------------------------------------------------------------
    static uint8_t Checksum8(const void* pvSrc_, uint16_t u16Len_);
```

```

    //---------------------------------------------------------------------------
    static uint16_t Checksum16(const void* pvSrc_, uint16_t u16Len_);

    //---------------------------------------------------------------------------
    static uint16_t StringLength(const char* szStr_);

    //---------------------------------------------------------------------------
    static bool CompareStrings(const char* szStr1_, const char* szStr2_);
    static bool CompareStrings(const char* szStr1_, const char* szStr2_, uint16_t u16Length_);

    //---------------------------------------------------------------------------
    static void CopyMemory(void* pvDst_, const void* pvSrc_, uint16_t u16Len_);

    //---------------------------------------------------------------------------
    static void CopyString(char* szDst_, const char* szSrc_);

    //---------------------------------------------------------------------------
    static int16_t StringSearch(const char* szBuffer_, const char* szPattern_);

    //---------------------------------------------------------------------------
    static bool CompareMemory(const void* pvMem1_, const void* pvMem2_, uint16_t u16Len_);

    //---------------------------------------------------------------------------
    static void SetMemory(void* pvDst_, uint8_t u8Val_, uint16_t u16Len_);

    //---------------------------------------------------------------------------
    static uint8_t Tokenize(const char* szBuffer_, Token_t* pastTokens_, uint8_t u8MaxTokens_);
};
} // namespace Mark3
```

20.11 /home/moslevin/projects/m3-repo/kernel/lib/streamer/public/streamer.h File Reference

Thread/Interrupt-safe byte-based data streaming.

```
#include "kerneltypes.h"
#include "mark3.h"
```

Classes

- class Mark3::Streamer

  The Streamer class. This class implements a circular byte-buffer with thread and interrupt safe methods for writing-to and reading-from the buffer. Objects of this class type are designed to be shared between threads, or between threads and interrupts.

Namespaces

- Mark3

20.11.1 Detailed Description

Thread/Interrupt-safe byte-based data streaming.

Definition in file streamer.h.

20.12 streamer.h

```
/*===========================================================================
     _____     _____     _____     _____
 ___|    _|__ __|_    |__ __|__   |__ __| __  |__  ______
|    \  /  | ||    \    ||     |     ||  |/ /    ||___   |
|     \/   | ||     \   ||     \     ||     \    ||___   |
|__/\__/|__|_||__|\__\ __||__|\__\ __||__|\__\ __||______|
    |_____|      |_____|     |_____|     |_____|

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

Copyright (c) 2012 - 2019 m0slevin, all rights reserved.
See license.txt for more information
=========================================================================== */
#include "kerneltypes.h"
#include "mark3.h"

#pragma once

namespace Mark3
{
//---------------------------------------------------------------------------
class Streamer
{
public:
    void Init(uint8_t* pau8Buffer_, uint16_t u16Size_);

    bool Read(uint8_t* pu8Data_);

    uint16_t Read(uint8_t* pu8Data_, uint16_t u16Len_);

    bool Write(uint8_t u8Data_);

    uint16_t Write(uint8_t* pu8Data_, uint16_t u16Len_);

    bool Claim(uint8_t** pu8Addr_);

    void Lock(uint8_t* pu8LockAddr_);

    void Unlock(void);

    uint16_t GetAvailable(void) { return m_u16Size; }
    bool CanRead(void);

    bool CanWrite(void);

    bool IsEmpty(void);

private:
    uint8_t* m_pau8Buffer;
    uint8_t* m_pu8LockAddr;

    uint16_t m_u16Size;
    uint16_t m_u16Avail;
    uint16_t m_u16Head;
    uint16_t m_u16Tail;
};
} // namespace Mark3
```

20.13 /home/moslevin/projects/m3-repo/kernel/lib/streamer/streamer.cpp File Reference

Thread/Interrupt-safe byte-based data streaming.

```
#include "kerneltypes.h"
#include "mark3.h"
#include "streamer.h"
```

Namespaces

- Mark3

20.13.1 Detailed Description

Thread/Interrupt-safe byte-based data streaming.

Definition in file streamer.cpp.

20.14 streamer.cpp

```
/*===========================================================================
     _____        _____        _____        _____
 ___|    _|__  __|_    |__  __|__   |__  __| __  |__  ______
|    \  /  | ||    \      ||     |     ||  |/ /    ||___   |
|     \/   | ||     \     ||     \     ||     \    ||___   |
|__/\__/|__|_||__|\__\  __||__|\__\  __||__|\__\ __||______|
    |_____|      |_____|     |_____|     |_____|

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

Copyright (c) 2012 - 2019 m0slevin, all rights reserved.
See license.txt for more information
=========================================================================== */
#include "kerneltypes.h"
#include "mark3.h"
#include "streamer.h"

namespace Mark3
{
//---------------------------------------------------------------------------
void Streamer::Init(uint8_t* pau8Buffer_, uint16_t u16Size_)
{
    m_u16Head      = 0;
    m_u16Tail      = 0;
    m_u16Size      = u16Size_;
    m_u16Avail     = m_u16Size;
    m_pau8Buffer   = pau8Buffer_;
    m_pu8LockAddr = 0;
}

//---------------------------------------------------------------------------
bool Streamer::Read(uint8_t* pu8Data_)
{
    auto rc = true;

    const auto cs = CriticalGuard{};

    if (m_u16Avail == m_u16Size) {
        rc = false;
    } else {
        auto* pu8Dest = &m_pau8Buffer[m_u16Tail];
        if (pu8Dest == m_pu8LockAddr) {
            rc = false;
        } else {
            *pu8Data_ = *pu8Dest;
            m_u16Tail++;
            if (m_u16Tail >= m_u16Size) {
                m_u16Tail = 0;
            }
            m_u16Avail++;
        }
    }

    return rc;
}

//---------------------------------------------------------------------------
uint16_t Streamer::Read(uint8_t* pu8Data_, uint16_t u16Len_)
{
    uint16_t u16ToRead;

    if (m_pu8LockAddr != 0) {
        return 0;
    }

    uint16_t u16Allocated;
    uint16_t u16PreWrap;
    uint8_t* pu8Src;
    uint8_t* pu8Dst;

    { // Begin critical section
```

```
        const auto cs = CriticalGuard{};
        u16Allocated = m_u16Size - m_u16Avail;

        if (u16Allocated > u16Len_) {
            u16ToRead = u16Len_;
        } else {
            u16ToRead = u16Allocated;
        }

        u16PreWrap = m_u16Size - m_u16Tail;

        pu8Src = &m_pau8Buffer[m_u16Tail];
        pu8Dst = pu8Data_;

        Lock(pu8Src);
    } // end critical section

    if (u16Allocated != 0u) {
        if (u16PreWrap >= u16ToRead) {
            for (uint16_t i = 0; i < u16ToRead; i++) { *pu8Dst++ = *pu8Src++; }
        } else {
            for (uint16_t i = 0; i < u16PreWrap; i++) { *pu8Dst++ = *pu8Src++; }
            pu8Src = m_pau8Buffer;
            for (uint16_t i = u16PreWrap; i < u16ToRead; i++) { *pu8Dst++ = *pu8Src++; }
        }
    }

    { // Begin critical section
        const auto cs = CriticalGuard{};
        m_u16Avail += u16ToRead;
        if (u16PreWrap >= u16ToRead) {
            m_u16Tail += u16ToRead;
        } else {
            m_u16Tail += u16ToRead - m_u16Size;
        }

        Unlock();
    } // end critical section
    return u16ToRead;
}

//---------------------------------------------------------------------------
bool Streamer::Write(uint8_t u8Data_)
{
    auto rc = true;

    const auto cs = CriticalGuard{};
    if (m_u16Avail == 0u) {
        rc = false;
    } else {
        if (m_pu8LockAddr == &m_pau8Buffer[m_u16Head]) {
            rc = false;
        } else {
            m_pau8Buffer[m_u16Head] = u8Data_;
            m_u16Head++;
            if (m_u16Head >= m_u16Size) {
                m_u16Head = 0;
            }
            m_u16Avail--;
        }
    }

    return rc;
}

//---------------------------------------------------------------------------
uint16_t Streamer::Write(uint8_t* pu8Data_, uint16_t u16Len_)
{
    uint16_t u16ToWrite;

    // Bail if the buffer is currently locked.
    if (m_pu8LockAddr != 0) {
        return 0;
    }

    // Update the buffer metadata in a critical section, and lock it so that
    // we can safely write to it with interrupts enabled.

    uint16_t u16PreWrap;
    uint8_t* pu8Src;
    uint8_t* pu8Dst;

    { // Begin critical section
        const auto cs = CriticalGuard{};
        if (m_u16Avail > u16Len_) {
            u16ToWrite = u16Len_;
        } else {
```

```
            u16ToWrite = m_u16Avail;
        }

        u16PreWrap = m_u16Size - m_u16Head;

        pu8Src = pu8Data_;
        pu8Dst = &m_pau8Buffer[m_u16Head];

        m_u16Avail -= u16ToWrite;

        if (u16PreWrap >= u16ToWrite) {
            m_u16Head += u16ToWrite;
        } else {
            m_u16Head += u16ToWrite - m_u16Size;
        }

        Lock(pu8Dst);
    } // End critical section

    // Perform the buffer writes with interrupts enabled, buffers locked.
    if (u16ToWrite != 0u) {
        if (u16PreWrap >= u16ToWrite) {
            for (uint16_t i = 0; i < u16ToWrite; i++) { *pu8Dst++ = *pu8Src++; }
        } else {
            for (uint16_t i = 0; i < u16PreWrap; i++) { *pu8Dst++ = *pu8Src++; }
            pu8Dst = m_pau8Buffer;
            for (uint16_t i = u16PreWrap; i < u16ToWrite; i++) { *pu8Dst++ = *pu8Src++; }
        }
    }

    Unlock();
    return u16ToWrite;
}

//---------------------------------------------------------------------------
bool Streamer::CanRead(void)
{
    auto bRc = true;
    const auto cs = CriticalGuard{};
    if (m_u16Avail == m_u16Size) {
        bRc = false;
    }
    return bRc;
}

//---------------------------------------------------------------------------
bool Streamer::CanWrite(void)
{
    auto bRc = false;

    const auto cs = CriticalGuard{};
    if (m_u16Avail != 0u) {
        bRc = true;
    }

    return bRc;
}

//---------------------------------------------------------------------------
bool Streamer::Claim(uint8_t** pu8Addr_)
{
    auto rc = true;

    const auto cs = CriticalGuard{};
    if (m_u16Avail == 0u) {
        rc = false;
    } else {
        if (m_pu8LockAddr == &m_pau8Buffer[m_u16Head]) {
            rc = false;
        } else {
            *pu8Addr_ = &m_pau8Buffer[m_u16Head];
            if (m_pu8LockAddr == 0) {
                m_pu8LockAddr = &m_pau8Buffer[m_u16Head];
            }
            m_u16Head++;
            if (m_u16Head >= m_u16Size) {
                m_u16Head = 0;
            }
            m_u16Avail--;
        }
    }
    return rc;
}

//---------------------------------------------------------------------------
void Streamer::Lock(uint8_t* pu8LockAddr_)
{
```

```
    const auto cs = CriticalGuard{};
    m_pu8LockAddr = pu8LockAddr_;
}

//---------------------------------------------------------------------------
void Streamer::Unlock(void)
{
    const auto cs = CriticalGuard{};
    m_pu8LockAddr = 0;
}

//---------------------------------------------------------------------------
bool Streamer::IsEmpty(void)
{
    const auto cs = CriticalGuard{};
    return m_u16Avail == m_u16Size;
}
} // namespace Mark3
```

20.15 /home/moslevin/projects/m3-repo/kernel/src/arch/avr/atmega1284p/gcc/kernelswi.cpp File Reference

Kernel Software interrupt implementation for ATMega1284p.

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

Namespaces

- Mark3

20.15.1 Detailed Description

Kernel Software interrupt implementation for ATMega1284p.

Definition in file kernelswi.cpp.

20.16 kernelswi.cpp

```
/*===========================================================================
     _____     _____     _____     _____
 ___|    _|__ __|_    |__ __|__   |__ __| __  |__ _____
|    \  /  | ||    \    ||     |    ||  |/ /    ||___   |
|     \/   | ||     \   ||     \    ||     \    ||___   |
|__/\__/|__|_||__|\__\ __||__|\__\ __||__|\__\ __||_____|
    |_____|      |_____|      |_____|      |_____|

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

Copyright (c) 2012 - 2019 m0slevin, all rights reserved.
See license.txt for more information
===========================================================================*/
#include "kerneltypes.h"
#include "kernelswi.h"

#include <avr/io.h>
#include <avr/interrupt.h>

namespace Mark3
```

```
{
//---------------------------------------------------------------------------
void KernelSWI::Config(void)
{
    PORTB &= ~0x04;                          // Clear INT2
    DDRB |= 0x04;                            // Set PortB, bit 2 (INT2) As Output
    EICRA |= (1 << ISC20) | (1 << ISC21); // Rising edge on INT2
}

//---------------------------------------------------------------------------
void KernelSWI::Start(void)
{
    EIFR &= ~(1 << INTF2); // Clear any pending interrupts on INT2
    EIMSK |= (1 << INT2);  // Enable INT2 interrupt (as int32_t as I-bit is set)
}

//---------------------------------------------------------------------------
void KernelSWI::Trigger(void)
{
    // if(Thread_IsSchedulerEnabled())
    {
        PORTB &= ~0x04;
        PORTB |= 0x04;
    }
}
} // namespace Mark3
```

20.17 /home/moslevin/projects/m3-repo/kernel/src/arch/avr/atmega1284p/gcc/kerneltimer.cpp File Reference

Kernel Timer Implementation for ATMega1284p.

```
#include "mark3.h"
#include <avr/common.h>
#include <avr/io.h>
#include <avr/interrupt.h>
```

Namespaces

- Mark3

Macros

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

Functions

- static void Mark3::KernelTimer_Task (void *unused)
- ISR (TIMER1_COMPA_vect)

20.17.1 Detailed Description

Kernel Timer Implementation for ATMega1284p.

Definition in file kerneltimer.cpp.

20.17.2 Macro Definition Documentation

20.17.2.1 TCCR1B_INIT

```
#define TCCR1B_INIT ((1 << WGM12) | (1 << CS12))
```

Definition at line 27 of file kerneltimer.cpp.

20.17.2.2 TIMER_IFR

```
#define TIMER_IFR (1 << OCF1A)
```

Definition at line 29 of file kerneltimer.cpp.

20.17.2.3 TIMER_IMSK

```
#define TIMER_IMSK (1 << OCIE1A)
```

Definition at line 28 of file kerneltimer.cpp.

20.17.3 Function Documentation

20.17.3.1 ISR()

```
ISR (
            TIMER1_COMPA_vect  )
```

Definition at line 103 of file kerneltimer.cpp.

20.18 kerneltimer.cpp

```
/*===========================================================================
     _____        _____        _____        _____
 ___|    _|__  __|_    |__  __|__   |__  __| __  |__  ______
|    \  /  | ||    \      ||     |     ||  |/ /     ||___   |
|     \/   | ||     \     ||     \     ||     \     ||___   |
|__/\__/|__|_||__|\__\  __||__|\__\  __||__|\__\  __||______|
    |_____|      |_____|      |_____|      |_____|

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

Copyright (c) 2012 - 2019 m0slevin, all rights reserved.
See license.txt for more information
===========================================================================*/
#include "mark3.h"

#include <avr/common.h>
#include <avr/io.h>
#include <avr/interrupt.h>

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

namespace
{
using namespace Mark3;
//---------------------------------------------------------------------------
// Static objects implementing the timer thread and its synchronization objects
Thread    s_clTimerThread;
K_WORD    s_clTimerThreadStack[PORT_KERNEL_TIMERS_THREAD_STACK];
Semaphore s_clTimerSemaphore;
}

namespace Mark3
{
//---------------------------------------------------------------------------
static void KernelTimer_Task(void* unused)
{
    (void)unused;
    while (1) {
        s_clTimerSemaphore.Pend();
#if KERNEL_ROUND_ROBIN
        Quantum::SetInTimer();
#endif // #if KERNEL_ROUND_ROBIN
        TimerScheduler::Process();
#if KERNEL_ROUND_ROBIN
        Quantum::ClearInTimer();
#endif // #if KERNEL_ROUND_ROBIN
    }
}

//---------------------------------------------------------------------------
void KernelTimer::Config(void)
{
    TCCR1B = TCCR1B_INIT;
    s_clTimerSemaphore.Init(0, 1);
    s_clTimerThread.Init(s_clTimerThreadStack,
                         sizeof(s_clTimerThreadStack) / sizeof(K_WORD),
                         KERNEL_TIMERS_THREAD_PRIORITY,
                         KernelTimer_Task,
                         0);
#if KERNEL_ROUND_ROBIN
    Quantum::SetTimerThread(&s_clTimerThread);
#endif // #if KERNEL_ROUND_ROBIN
    s_clTimerThread.Start();
}

//---------------------------------------------------------------------------
void KernelTimer::Start(void)
{
    TCCR1B = ((1 << WGM12) | (1 << CS11) | (1 << CS10));
    OCR1A  = ((PORT_SYSTEM_FREQ / 1000) / 64);
    TCNT1  = 0;
    TIFR1 &= ~TIMER_IFR;
    TIMSK1 |= TIMER_IMSK;
}

//---------------------------------------------------------------------------
void KernelTimer::Stop(void)
{
    TIFR1 &= ~TIMER_IFR;
    TIMSK1 &= ~TIMER_IMSK;
    TCCR1B &= ~(1 << CS12); // Disable count...
    TCNT1 = 0;
```

```
    OCR1A = 0;
}
} // namespace Mark3

//---------------------------------------------------------------------------
//---------------------------------------------------------------------------
using namespace Mark3;
ISR(TIMER1_COMPA_vect)
{
    Kernel::Tick();
    s_clTimerSemaphore.Post();
}
```

20.19 /home/moslevin/projects/m3-repo/kernel/src/arch/avr/atmega1284p/gcc/public/portcfg.h File Reference

Mark3 Port Configuration.

Macros

- #define KERNEL_NUM_PRIORITIES (16)
- #define KERNEL_TIMERS_THREAD_PRIORITY (KERNEL_NUM_PRIORITIES - 1)
- #define THREAD_QUANTUM_DEFAULT (4)
- #define KERNEL_STACK_GUARD_DEFAULT (32)
- #define PORT_COROUTINE_PRIORITIES (8)
- #define AVR (1)
- #define K_WORD uint8_t

    Size of a data word.
- #define K_ADDR uint16_t

    Size of an address (pointer size)
- #define K_INT int32_t
- #define PORT_PRIO_TYPE uint8_t

    Type used for bitmap in the PriorityMap class.
- #define PORT_PRIO_MAP_WORD_SIZE (1)

    size of PORT_PRIO_TYPE in bytes
- #define PORT_SYSTEM_FREQ ((uint32_t)16000000)

    CPU Frequency in Hz.
- #define PORT_TIMER_FREQ ((uint32_t)(PORT_SYSTEM_FREQ / 1000))

    Fixed timer interrupt frequency.
- #define PORT_KERNEL_DEFAULT_STACK_SIZE ((K_ADDR)384)
- #define PORT_KERNEL_TIMERS_THREAD_STACK ((K_ADDR)384)
- #define PORT_TIMER_COUNT_TYPE uint16_t

    Timer counter type.
- #define PORT_MIN_TIMER_TICKS (0)
- #define PORT_OVERLOAD_NEW (1)
- #define PORT_STACK_GROWS_DOWN (1)
- #define PORT_USE_HW_CLZ (1)

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 AVR

```
#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 55 of file portcfg.h.

20.19.2.2 K_ADDR

```
#define K_ADDR uint16_t
```

Size of an address (pointer size)

Examples:

lab9_dynamic_threads/main.cpp.

Definition at line 63 of file portcfg.h.

20.19.2.3 K_INT

```
#define K_INT int32_t
```

Definition at line 64 of file portcfg.h.

20.19.2.4 K_WORD

```
#define K_WORD uint8_t
```

Size of a data word.

Define types that map to the CPU Architecture's default data-word and address size.

Examples:

lab10_notifications/main.cpp, lab11_mailboxes/main.cpp, lab1_kernel_setup/main.cpp, lab2_coroutines/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 62 of file portcfg.h.

20.19.2.5 KERNEL_NUM_PRIORITIES

```
#define KERNEL_NUM_PRIORITIES (16)
```

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 35 of file portcfg.h.

20.19.2.6 KERNEL_STACK_GUARD_DEFAULT

```
#define KERNEL_STACK_GUARD_DEFAULT (32)
```

Definition at line 41 of file portcfg.h.

20.19.2.7 KERNEL_TIMERS_THREAD_PRIORITY

```
#define KERNEL_TIMERS_THREAD_PRIORITY (KERNEL_NUM_PRIORITIES - 1)
```

Definition at line 37 of file portcfg.h.

20.19.2.8 PORT_COROUTINE_PRIORITIES

```
#define PORT_COROUTINE_PRIORITIES (8)
```

Set the number of priorities supported by the coroutine scheduler. The number of coroutine priorities is limited by the memory of the host CPU.

Definition at line 47 of file portcfg.h.

20.19.2.9 PORT_KERNEL_DEFAULT_STACK_SIZE

```
#define PORT_KERNEL_DEFAULT_STACK_SIZE ((K_ADDR)384)
```

Define the default/minimum size of a thread stack

Examples:

lab10_notifications/main.cpp, lab11_mailboxes/main.cpp, lab1_kernel_setup/main.cpp, lab2_coroutines/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 96 of file portcfg.h.

20.19.2.10 PORT_KERNEL_TIMERS_THREAD_STACK

```
#define PORT_KERNEL_TIMERS_THREAD_STACK ((K_ADDR)384)
```

Define the size of the kernel-timer thread stack (if one is configured)

Definition at line 101 of file portcfg.h.

20.19.2.11 PORT_MIN_TIMER_TICKS

```
#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.

Definition at line 112 of file portcfg.h.

20.19.2.12 PORT_OVERLOAD_NEW

```
#define PORT_OVERLOAD_NEW (1)
```

Set this to 1 to overload the system's New/Free functions with the kernel's allocator functions. A user must configure the Kernel's allocator functions to point to a real heap implementation backed with real memory in order to use dynamic object creation.

Definition at line 119 of file portcfg.h.

20.19.2.13 PORT_PRIO_MAP_WORD_SIZE

```
#define PORT_PRIO_MAP_WORD_SIZE (1)
```

size of PORT_PRIO_TYPE in bytes

Definition at line 74 of file portcfg.h.

20.19.2.14 PORT_PRIO_TYPE

```
#define PORT_PRIO_TYPE uint8_t
```

Type used for bitmap in the PriorityMap class.

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.

Definition at line 73 of file portcfg.h.

20.19.2.15 PORT_STACK_GROWS_DOWN

```
#define PORT_STACK_GROWS_DOWN (1)
```

Set this to 1 if the stack grows down in the target architecture, or 0 if the stack grows up

Definition at line 124 of file portcfg.h.

20.19.2.16 PORT_SYSTEM_FREQ

```
#define PORT_SYSTEM_FREQ ((uint32_t)16000000)
```

CPU Frequency in Hz.

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.

Definition at line 81 of file portcfg.h.

20.19.2.17 PORT_TIMER_COUNT_TYPE

```
#define PORT_TIMER_COUNT_TYPE uint16_t
```

Timer counter type.

Define the native type corresponding to the kernel timer hardware's counter register.

Definition at line 106 of file portcfg.h.

20.19.2.18 PORT_TIMER_FREQ

```
#define PORT_TIMER_FREQ ((uint32_t)(PORT_SYSTEM_FREQ / 1000))
```

Fixed timer interrupt frequency.

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.

Definition at line 91 of file portcfg.h.

20.19.2.19 PORT_USE_HW_CLZ

```
#define PORT_USE_HW_CLZ (1)
```

Set this to 1 if the target CPU/toolchain supports an optimized Count-leading-zeros instruction, or count-leading-zeros intrinsic. If such functionality is not available, a general-purpose implementation will be used.

Definition at line 131 of file portcfg.h.

20.19.2.20 THREAD_QUANTUM_DEFAULT

```
#define THREAD_QUANTUM_DEFAULT (4)
```

Definition at line 39 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 - 2019 m0slevin, all rights reserved.
00012 See license.txt for more information
00013 ===========================================================================*/
00024 #pragma once
00025
00035 #define KERNEL_NUM_PRIORITIES (16)
00036
00037 #define KERNEL_TIMERS_THREAD_PRIORITY (KERNEL_NUM_PRIORITIES - 1)
00038
00039 #define THREAD_QUANTUM_DEFAULT (4)
00040
00041 #define KERNEL_STACK_GUARD_DEFAULT (32) // words
00042
00047 #define PORT_COROUTINE_PRIORITIES (8)
00048
00054 #ifndef AVR
00055 #define AVR (1)
00056 #endif
00057
00062 #define K_WORD uint8_t
00063 #define K_ADDR uint16_t
00064 #define K_INT int32_t
00065
00073 #define PORT_PRIO_TYPE uint8_t
00074 #define PORT_PRIO_MAP_WORD_SIZE (1)
00075
00076
00080 #if !defined(PORT_SYSTEM_FREQ)
00081 #define PORT_SYSTEM_FREQ ((uint32_t)16000000)
00082 #endif
00083
00091 #define PORT_TIMER_FREQ ((uint32_t)(PORT_SYSTEM_FREQ / 1000))
00092
00093
00096 #define PORT_KERNEL_DEFAULT_STACK_SIZE ((K_ADDR)384)
00097
00101 #define PORT_KERNEL_TIMERS_THREAD_STACK ((K_ADDR)384)
00102
00106 #define PORT_TIMER_COUNT_TYPE uint16_t
00107
00108
00112 #define PORT_MIN_TIMER_TICKS (0)
00113
00119 #define PORT_OVERLOAD_NEW (1)
00120
00124 #define PORT_STACK_GROWS_DOWN (1)
00125
00131 #define PORT_USE_HW_CLZ   (1)
```

20.21 /home/moslevin/projects/m3-repo/kernel/src/arch/avr/atmega1284p/gcc/public/threadport.h File Reference

ATMega1284p Multithreading support.

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

Namespaces

- Mark3

Macros

- #define ASM(x) asm volatile(x);
- #define PORT_TOP_OF_STACK(x, y) (reinterpret_cast<K_WORD∗>(reinterpret_cast<K_ADDR>(x) + (static_cast<K_ADDR>(y) - 1)))

    Macro to find the top of a stack given its size and top address.
- #define PORT_PUSH_TO_STACK(x, y) ∗x = y; x--;

    Push a value y to the stack pointer x and decrement the stack pointer.
- #define Thread_SaveContext()

    Save the context of the Thread.
- #define Thread_RestoreContext()

    Restore the context of the Thread.

Functions

- uint8_t Mark3::PORT_CLZ (uint8_t in_)
- void Mark3::PORT_IRQ_ENABLE ()
- void Mark3::PORT_IRQ_DISABLE ()
- void Mark3::PORT_CS_ENTER ()
- void Mark3::PORT_CS_EXIT ()
- K_WORD Mark3::PORT_CS_NESTING ()

Variables

- static constexpr auto Mark3::SR_ = uint8_t{0x3F}
- K_WORD Mark3::g_kwSFR
- K_WORD Mark3::g_kwCriticalCount

20.21.1 Detailed Description

ATMega1284p Multithreading support.

Definition in file threadport.h.

20.21.2 Macro Definition Documentation

20.21.2.1 ASM

```
#define ASM(
              x ) asm volatile(x);
```

Definition at line 32 of file threadport.h.

20.21.2.2 PORT_PUSH_TO_STACK

```
#define PORT_PUSH_TO_STACK(
              x,
              y ) *x = y; x--;
```

Push a value y to the stack pointer x and decrement the stack pointer.

Definition at line 38 of file threadport.h.

20.21.2.3 PORT_TOP_OF_STACK

```
#define PORT_TOP_OF_STACK(
              x,
              y ) (reinterpret_cast<K_WORD*>(reinterpret_cast<K_ADDR>(x) + (static_cast<K_A←
DDR>(y) - 1)))
```

Macro to find the top of a stack given its size and top address.

Definition at line 36 of file threadport.h.

20.21.2.4 Thread_RestoreContext

```
#define Thread_RestoreContext( )
```

Restore the context of the Thread.

Definition at line 91 of file threadport.h.

20.21.2.5 Thread_SaveContext

```
#define Thread_SaveContext( )
```

Save the context of the Thread.

Definition at line 42 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 - 2019 m0slevin, all rights reserved.
00012 See license.txt for more information
00013 ========================================================================== */
00020 #pragma once
00021
00022 #include "portcfg.h"
00023 #include "kerneltypes.h"
00024
00025 #include <avr/io.h>
00026 #include <avr/interrupt.h>
00027
00028 namespace Mark3
00029 {
00030 // clang-format off
00031 //---------------------------------------------------------------------------
00032 #define ASM(x)      asm volatile(x);
00033
00034 //---------------------------------------------------------------------------
00036 #define PORT_TOP_OF_STACK(x, y)    (reinterpret_cast<K_WORD*>(reinterpret_cast<K_ADDR>(x) +
       (static_cast<K_ADDR>(y) - 1)))
00037 #define PORT_PUSH_TO_STACK(x, y)        *x = y; x--;
00039
00040 //---------------------------------------------------------------------------
00042 #define Thread_SaveContext() \
00043 ASM("push r0"); \
00044 ASM("in r0, __SREG__"); \
00045 ASM("cli"); \
00046 ASM("push r0"); \
00047 ASM("push r1"); \
00048 ASM("clr r1"); \
00049 ASM("push r2"); \
00050 ASM("push r3"); \
00051 ASM("push r4"); \
00052 ASM("push r5"); \
00053 ASM("push r6"); \
00054 ASM("push r7"); \
00055 ASM("push r8"); \
00056 ASM("push r9"); \
00057 ASM("push r10"); \
00058 ASM("push r11"); \
00059 ASM("push r12"); \
00060 ASM("push r13"); \
00061 ASM("push r14"); \
00062 ASM("push r15"); \
00063 ASM("push r16"); \
00064 ASM("push r17"); \
00065 ASM("push r18"); \
00066 ASM("push r19"); \
00067 ASM("push r20"); \
00068 ASM("push r21"); \
00069 ASM("push r22"); \
00070 ASM("push r23"); \
00071 ASM("push r24"); \
00072 ASM("push r25"); \
00073 ASM("push r26"); \
00074 ASM("push r27"); \
00075 ASM("push r28"); \
00076 ASM("push r29"); \
```

```
ASM("push r30"); \
ASM("push r31"); \
ASM("in    r0, 0x3B"); \
ASM("push r0"); \
ASM("lds r26, g_pclCurrent"); \
ASM("lds r27, g_pclCurrent + 1"); \
ASM("adiw r26, 4"); \
ASM("in    r0, 0x3D"); \
ASM("st    x+, r0"); \
ASM("in    r0, 0x3E"); \
ASM("st    x+, r0");

//---------------------------------------------------------------------------
#define Thread_RestoreContext() \
ASM("lds r26, g_pclCurrent"); \
ASM("lds r27, g_pclCurrent + 1");\
ASM("adiw r26, 4"); \
ASM("ld     r28, x+"); \
ASM("out 0x3D, r28"); \
ASM("ld     r29, x+"); \
ASM("out 0x3E, r29"); \
ASM("pop r0"); \
ASM("out 0x3B, r0"); \
ASM("pop r31"); \
ASM("pop r30"); \
ASM("pop r29"); \
ASM("pop r28"); \
ASM("pop r27"); \
ASM("pop r26"); \
ASM("pop r25"); \
ASM("pop r24"); \
ASM("pop r23"); \
ASM("pop r22"); \
ASM("pop r21"); \
ASM("pop r20"); \
ASM("pop r19"); \
ASM("pop r18"); \
ASM("pop r17"); \
ASM("pop r16"); \
ASM("pop r15"); \
ASM("pop r14"); \
ASM("pop r13"); \
ASM("pop r12"); \
ASM("pop r11"); \
ASM("pop r10"); \
ASM("pop r9"); \
ASM("pop r8"); \
ASM("pop r7"); \
ASM("pop r6"); \
ASM("pop r5"); \
ASM("pop r4"); \
ASM("pop r3"); \
ASM("pop r2"); \
ASM("pop r1"); \
ASM("pop r0"); \
ASM("out __SREG__, r0"); \
ASM("pop r0");

//---------------------------------------------------------------------------
static constexpr auto SR_ = uint8_t{0x3F};
extern "C" {
   extern K_WORD g_kwSFR;
   extern K_WORD g_kwCriticalCount;
}

//---------------------------------------------------------------------------
inline uint8_t PORT_CLZ(uint8_t in_)
{
    static const uint8_t u8Lookup[] = {4, 3, 2, 2, 1, 1, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0};
    uint8_t hi = __builtin_avr_swap(in_) & 0x0F;
    if (hi) {
        return u8Lookup[hi];
    }
    return 4 + u8Lookup[in_];
}

//---------------------------------------------------------------------------
inline void PORT_IRQ_ENABLE()
{
    ASM("sei");
}

//---------------------------------------------------------------------------
inline void PORT_IRQ_DISABLE()
{
    ASM("cli");
}
```

```

//---------------------------------------------------------------------------
inline void PORT_CS_ENTER()
{
    auto u8SFR = _SFR_IO8(SR_);
    ASM("cli");
    if (!g_kwCriticalCount) {
        g_kwSFR = u8SFR;
    }
    g_kwCriticalCount++;
}

//---------------------------------------------------------------------------
inline void PORT_CS_EXIT()
{
    g_kwCriticalCount--;
    if (!g_kwCriticalCount) {
        _SFR_IO8(SR_) = g_kwSFR;
    }
}

//---------------------------------------------------------------------------
inline K_WORD PORT_CS_NESTING()
{
    return g_kwCriticalCount;
}
} // namespace Mark3
```

20.23 /home/moslevin/projects/m3-repo/kernel/src/arch/avr/atmega1284p/gcc/threadport.cpp File Reference

ATMega1284p 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 <avr/io.h>
#include <avr/interrupt.h>
```

Namespaces

- Mark3

Functions

- static void Mark3::Thread_Switch (void)
- Mark3::ISR (INT2_vect) __attribute__((signal

  ISR(INT2_vect) SWI using INT2 - used to trigger a context switch.

Variables

- K_WORD g_kwSFR = 0
- K_WORD g_kwCriticalCount = 0
- Mark3::naked

20.23.1 Detailed Description

ATMega1284p Multithreading.

Definition in file threadport.cpp.

20.23.2 Variable Documentation

20.23.2.1 g_kwCriticalCount

```
K_WORD g_kwCriticalCount = 0
```

Definition at line 36 of file threadport.cpp.

20.23.2.2 g_kwSFR

```
K_WORD g_kwSFR = 0
```

Definition at line 35 of file threadport.cpp.

20.24 threadport.cpp

```
/*===========================================================================
     _____        _____        _____        _____
 ___|    _|__  __|_    |__  __|__   |__  __| __  |__  ______
|    \  /  | ||    \      ||     |     ||  |/ /     ||___   |
|     \/   | ||     \     ||     \     ||     \     ||___   |
|__/\__/|__|_||__|\__\  __||__|\__\  __||__|\__\  __||______|
    |_____|      |_____|      |_____|      |_____|

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

Copyright (c) 2012 - 2019 m0slevin, all rights reserved.
See license.txt for more information
===========================================================================*/
#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 <avr/io.h>
#include <avr/interrupt.h>

extern "C" {
K_WORD g_kwSFR = 0;
K_WORD g_kwCriticalCount = 0;
} // extern "C"

//---------------------------------------------------------------------------
namespace Mark3
{
//---------------------------------------------------------------------------
void ThreadPort::InitStack(Thread* pclThread_)
{
```

```
    // Initialize the stack for a Thread
    uint16_t u16Addr;
    uint8_t* pu8Stack;
    uint16_t i;

    // Get the address of the thread's entry function
    u16Addr = (uint16_t)(pclThread_->m_pfEntryPoint);

    // Start by finding the bottom of the stack
    pu8Stack = (uint8_t*)pclThread_->m_pwStackTop;

#if KERNEL_STACK_CHECK
    // clear the stack, and initialize it to a known-default value (easier
    // to debug when things go sour with stack corruption or overflow)
    for (i = 0; i < pclThread_->m_u16StackSize; i++) { pclThread_->m_pwStack[i] = 0xFF; }
#endif // #if KERNEL_STACK_CHECK

    // Our context starts with the entry function
    PORT_PUSH_TO_STACK(pu8Stack, (uint8_t)(u16Addr & 0x00FF));
    PORT_PUSH_TO_STACK(pu8Stack, (uint8_t)((u16Addr >> 8) & 0x00FF));

    // R0
    PORT_PUSH_TO_STACK(pu8Stack, 0x00); // R0

    // Push status register and R1 (which is used as a constant zero)
    PORT_PUSH_TO_STACK(pu8Stack, 0x80); // SR
    PORT_PUSH_TO_STACK(pu8Stack, 0x00); // R1

    // Push other registers
    for (i = 2; i <= 23; i++) // R2-R23
    {
        PORT_PUSH_TO_STACK(pu8Stack, i);
    }

    // Assume that the argument is the only stack variable
    PORT_PUSH_TO_STACK(pu8Stack, (uint8_t)(((uint16_t)(pclThread_->m_pvArg)) & 0x00FF));        // R24
    PORT_PUSH_TO_STACK(pu8Stack, (uint8_t)((((uint16_t)(pclThread_->m_pvArg)) >> 8) & 0x00FF)); // R25

    // Push the rest of the registers in the context
    for (i = 26; i <= 31; i++) { PORT_PUSH_TO_STACK(pu8Stack, i); }

    PORT_PUSH_TO_STACK(pu8Stack, 0x00); // RAMPZ
    // Set the top o' the stack.
    pclThread_->m_pwStackTop = (uint8_t*)pu8Stack;

    // That's it!  the thread is ready to run now.
}

//---------------------------------------------------------------------------
static void Thread_Switch(void)
{
    g_pclCurrent = (Thread*)g_pclNext;
}

//---------------------------------------------------------------------------
void ThreadPort::StartThreads()
{
    KernelSWI::Config();   // configure the task switch SWI
    KernelTimer::Config(); // configure the kernel timer

    // Tell the kernel that we're ready to start scheduling threads
    // for the first time.
    Kernel::CompleteStart();

    Scheduler::SetScheduler(1); // enable the scheduler
    Scheduler::Schedule();      // run the scheduler - determine the first thread to run

    Thread_Switch(); // Set the next scheduled thread to the current thread

    KernelTimer::Start(); // enable the kernel timer
    KernelSWI::Start();   // enable the task switch SWI

#if KERNEL_ROUND_ROBIN
    // Restart the thread quantum timer, as any value held prior to starting
    // the kernel will be invalid.  This fixes a bug where multiple threads
    // started with the highest priority before starting the kernel causes problems
    // until the running thread voluntarily blocks.
    Quantum::Update(g_pclCurrent);
#endif // #if KERNEL_ROUND_ROBIN

    // Restore the context...
    Thread_RestoreContext(); // restore the context of the first running thread
    ASM("reti");             // return from interrupt - will return to the first scheduled thread
}
```

```

//---------------------------------------------------------------------------
//---------------------------------------------------------------------------
ISR(INT2_vect) __attribute__((signal, naked));
ISR(INT2_vect)
{
    Thread_SaveContext();    // Push the context (registers) of the current task
    Thread_Switch();         // Switch to the next task
    Thread_RestoreContext(); // Pop the context (registers) of the next task
    ASM("reti");             // Return to the next task
}
} // namespace Mark3
```

20.25 /home/moslevin/projects/m3-repo/kernel/src/atomic.cpp File Reference

Basic Atomic Operations.

`#include "mark3.h"`

Namespaces

- Mark3

20.25.1 Detailed Description

Basic Atomic Operations.

Definition in file atomic.cpp.

20.26 atomic.cpp

```
/*===========================================================================
     _____        _____        _____        _____
 ___|    _|__  __|_    |__  __|__   |__  __| __  |__  ______
|    \  /  | ||    \      ||     |     ||  |/ /     ||___   |
|     \/   | ||     \     ||     \     ||     \     ||___   |
|__/\__/|__|_||__|\__\  __||__|\__\  __||__|\__\  __||______|
    |_____|      |_____|      |_____|      |_____|

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

Copyright (c) 2012 - 2019 m0slevin, all rights reserved.
See license.txt for more information
===========================================================================*/
#include "mark3.h"

namespace Mark3
{
//---------------------------------------------------------------------------
bool Atomic::TestAndSet(bool* pbLock_)
{
    KERNEL_ASSERT(nullptr != pbLock_);

    auto cs = CriticalGuard{};
    auto bRet = *pbLock_;
    if (!bRet) {
        *pbLock_ = 1;
    }
    return bRet;
}
} // namespace Mark3
```

20.27 /home/moslevin/projects/m3-repo/kernel/src/autoalloc.cpp File Reference

Automatic memory allocation for kernel objects.

```
#include "mark3.h"
#include <stdint.h>
```

Namespaces

- Mark3

20.27.1 Detailed Description

Automatic memory allocation for kernel objects.

Definition in file autoalloc.cpp.

20.28 autoalloc.cpp

```
/*===========================================================================
     _____     _____     _____     _____
 ___|    _|__ __|_    |__ __|__   |__ __| __  |__ ______
|    \  /  | ||    \    ||     |     ||  |/ /    ||___   |
|     \/   | ||     \   ||     \     ||     \    ||___   |
|__/\__/|__|_||__|\__\ __||__|\__\ __||__|\__\ __||______|
    |_____|      |_____|     |_____|     |_____|

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

Copyright (c) 2012 - 2019 m0slevin, all rights reserved.
See license.txt for more information
=========================================================================== */
#include "mark3.h"

#include <stdint.h>

//---------------------------------------------------------------------------
// Override new() and delete() using functions provided to AutoAlloc
//---------------------------------------------------------------------------
#if PORT_OVERLOAD_NEW
using namespace Mark3;
void* operator new(size_t n)
{
    return AutoAlloc::NewRawData(n);
}

//---------------------------------------------------------------------------
void* operator new[](size_t n)
{
    return AutoAlloc::NewRawData(n);
}

//---------------------------------------------------------------------------
void operator delete(void* p)
{
    AutoAlloc::DestroyRawData(p);
}

//---------------------------------------------------------------------------
void operator delete[](void* p)
{
    AutoAlloc::DestroyRawData(p);
}
#endif

namespace Mark3
{
```

```
AutoAllocAllocator_t AutoAlloc::m_pfAllocator;
AutoAllocFree_t      AutoAlloc::m_pfFree;

//---------------------------------------------------------------------------
void* AutoAlloc::Allocate(AutoAllocType eType_, size_t sSize_)
{
    if (!m_pfAllocator) {
        return nullptr;
    }
    return m_pfAllocator(eType_, sSize_);
}

//---------------------------------------------------------------------------
void AutoAlloc::Free(AutoAllocType eType_, void* pvObj_)
{
    if (!m_pfFree) {
        return;
    }
    m_pfFree(eType_, pvObj_);
}

//---------------------------------------------------------------------------
void AutoAlloc::SetAllocatorFunctions(
AutoAllocAllocator_t pfAllocator_, AutoAllocFree_t pfFree_)
{
    m_pfAllocator = pfAllocator_;
    m_pfFree      = pfFree_;
}

//---------------------------------------------------------------------------
void AutoAlloc::Init()
{
    m_pfAllocator = nullptr;
    m_pfFree      = nullptr;
}

//---------------------------------------------------------------------------
void* AutoAlloc::NewUserTypeAllocation(uint8_t eType_)
{
    return Allocate(static_cast<AutoAllocType>(eType_), 0);
}
//---------------------------------------------------------------------------
void AutoAlloc::DestroyUserTypeAllocation(uint8_t eUserType_, void*
pvObj_)
{
    Free(static_cast<AutoAllocType>(eUserType_), pvObj_);
}
//---------------------------------------------------------------------------
void* AutoAlloc::NewRawData(size_t sSize_)
{
    return Allocate(AutoAllocType::Raw, sSize_);
}
//---------------------------------------------------------------------------
void AutoAlloc::DestroyRawData(void* pvData_)
{
    Free(AutoAllocType::Raw, pvData_);
}

} // namespace Mark3
```

20.29 /home/moslevin/projects/m3-repo/kernel/src/blocking.cpp File Reference

Implementation of base class for blocking objects.

```
#include "mark3.h"
```

Namespaces

- Mark3

20.29.1 Detailed Description

Implementation of base class for blocking objects.

Definition in file blocking.cpp.

20.30 blocking.cpp

```
/*===========================================================================
     _____     _____     _____     _____
 ___|    _|__  __|_    |__  __|__   |__  __| __  |__  _____
|    \  /  | ||    \     ||     |     ||  |/ /     ||___   |
|     \/   | ||     \    ||     \     ||     \     ||___   |
|__/\__/|__|_||__|\__\  __||__|\__\  __||__|\__\  __||_______|
    |_____|      |_____|      |_____|      |_____|

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

Copyright (c) 2012 - 2019 m0slevin, all rights reserved.
See license.txt for more information
===========================================================================*/
#include "mark3.h"

namespace Mark3
{
//---------------------------------------------------------------------------
void BlockingObject::Block(Thread* pclThread_)
{
    KERNEL_ASSERT(nullptr != pclThread_);

    // Remove the thread from its current thread list (the "owner" list)
    // ... And add the thread to this object's block list
    Scheduler::Remove(pclThread_);
    m_clBlockList.Add(pclThread_);

    // Set the "current" list location to the blocklist for this thread
    pclThread_->SetCurrent(&m_clBlockList);
    pclThread_->SetState(ThreadState::Blocked);
}

//---------------------------------------------------------------------------
void BlockingObject::BlockPriority(Thread* pclThread_)
{
    KERNEL_ASSERT(nullptr != pclThread_);

    // Remove the thread from its current thread list (the "owner" list)
    // ... And add the thread to this object's block list
    Scheduler::Remove(pclThread_);
    m_clBlockList.AddPriority(pclThread_);

    // Set the "current" list location to the blocklist for this thread
    pclThread_->SetCurrent(&m_clBlockList);
    pclThread_->SetState(ThreadState::Blocked);
}

//---------------------------------------------------------------------------
void BlockingObject::UnBlock(Thread* pclThread_)
{
    KERNEL_ASSERT(nullptr != pclThread_);

    // Remove the thread from its current thread list (the "owner" list)
    pclThread_->GetCurrent()->Remove(pclThread_);

    // Put the thread back in its active owner's list.  This is usually
    // the ready-queue at the thread's original priority.
    Scheduler::Add(pclThread_);

    // Tag the thread's current list location to its owner
    pclThread_->SetCurrent(pclThread_->GetOwner());
    pclThread_->SetState(ThreadState::Ready);
}
} // namespace Mark3
```

20.31 /home/moslevin/projects/m3-repo/kernel/src/colist.cpp File Reference

CoRoutine List structure implementation.

```
#include "colist.h"
```

Namespaces

- Mark3

20.31.1 Detailed Description

CoRoutine List structure implementation.

Definition in file colist.cpp.

20.32 colist.cpp

```
/*===========================================================================
     _____     _____     _____     _____
 ___|    _|__  __|_    |__  __|__   |__  __| __  |__  ______
|    \  /  | ||    \      ||     |     ||  |/ /     ||___   |
|     \/   | ||     \     ||     \     ||  |\ \     ||___   |
|__/\__/|__|_||__|\__\  __||__|\__\  __||__|\__\  __||______|
    |_____|      |_____|     |_____|     |_____|

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

Copyright (c) 2012 - 2019 m0slevin, all rights reserved.
See license.txt for more information
=========================================================================== */
#include "colist.h"

namespace Mark3 {
//---------------------------------------------------------------------------
void CoList::SetPrioMap(CoPrioMap* pclPrioMap_)
{
    m_pclPrioMap = pclPrioMap_;
}

//---------------------------------------------------------------------------
void CoList::SetPriority(PORT_PRIO_TYPE uPriority)
{
    m_uPriority = uPriority;
}

//---------------------------------------------------------------------------
void CoList::Add(Coroutine* pclCoroutine_)
{
    TypedCircularLinkList<Coroutine>::Add(pclCoroutine_);
    PivotForward();
    if (m_pclPrioMap) {
        m_pclPrioMap->Set(m_uPriority);
    }
}

//---------------------------------------------------------------------------
void CoList::Remove(Coroutine* pclCoroutine_)
{
    TypedCircularLinkList<Coroutine>::Remove(pclCoroutine_);
    if (m_pclPrioMap) {
        if (nullptr == GetHead()) {
            m_pclPrioMap->Clear(m_uPriority);
        }
    }
}
} // namespace Mark3
```

20.33 /home/moslevin/projects/m3-repo/kernel/src/condvar.cpp File Reference

Condition Variable implementation.

```
#include "mark3.h"
```

Namespaces

- Mark3

20.33.1 Detailed Description

Condition Variable implementation.

Definition in file condvar.cpp.

20.34 condvar.cpp

```
/*===========================================================================
     _____        _____        _____        _____
 ___|    _|__  __|_    |__  __|__   |__  __| __  |__  ______
|    \  /  | ||    \      ||     |     ||  |/ /     ||___   |
|     \/   | ||     \     ||     \     ||     \     ||___   |
|__/\__/|__|_||__|\__\  __||__|\__\  __||__|\__\  __||______|
    |_____|      |_____|      |_____|      |_____|

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

Copyright (c) 2012 - 2018 m0slevin, all rights reserved.
See license.txt for more information
===========================================================================*/
#include "mark3.h"

namespace Mark3
{
//---------------------------------------------------------------------------
void ConditionVariable::Init()
{
    m_clMutex.Init();
    m_clSemaphore.Init(0, 255);
}

//---------------------------------------------------------------------------
void ConditionVariable::Wait(Mutex* pclMutex_)
{
    KERNEL_ASSERT(nullptr != pclMutex_);

    m_clMutex.Claim();

    pclMutex_->Release();
    m_u8Waiters++;

    m_clMutex.Release();

    m_clSemaphore.Pend();
    pclMutex_->Claim();
}

//---------------------------------------------------------------------------
bool ConditionVariable::Wait(Mutex* pclMutex_, uint32_t u32WaitTimeMS_)
{
    KERNEL_ASSERT(nullptr != pclMutex_);

    m_clMutex.Claim();

    pclMutex_->Release();
    m_u8Waiters++;
```

```

    m_clMutex.Release();

    if (!m_clSemaphore.Pend(u32WaitTimeMS_)) {
        return false;
    }
    return pclMutex_->Claim(u32WaitTimeMS_);
}

//---------------------------------------------------------------------------
void ConditionVariable::Signal()
{
    m_clMutex.Claim();
    if (m_u8Waiters) {
        m_u8Waiters--;
        m_clSemaphore.Post();
    }
    m_clMutex.Release();
}

//---------------------------------------------------------------------------
void ConditionVariable::Broadcast()
{
    m_clMutex.Claim();

    while (m_u8Waiters > 0) {
        m_u8Waiters--;
        m_clSemaphore.Post();
    }

    m_clMutex.Release();
}

} // namespace Mark3
```

20.35 /home/moslevin/projects/m3-repo/kernel/src/coroutine.cpp File Reference

Coroutine object implementation.

```
#include "coroutine.h"
#include "cosched.h"
#include "criticalguard.h"
#include "kernel.h"
```

Namespaces

- Mark3

20.35.1 Detailed Description

Coroutine object implementation.

Definition in file coroutine.cpp.

20.36 coroutine.cpp

```
/*===========================================================================
     _____        _____        _____        _____
 ___|    _|__  __|_    |__  __|__   |__  __| __  |__  ______
|    \  /  | ||    \      ||     |     ||  |/ /     ||___   |
|     \/   | ||     \     ||     \     ||     \     ||___   |
|__/\__/|__|_||__|\__\  __||__|\__\  __||__|\__\  __||______|
    |_____|      |_____|      |_____|      |_____|

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

Copyright (c) 2012 - 2019 m0slevin, all rights reserved.
See license.txt for more information
=========================================================================== */
#include "coroutine.h"
#include "cosched.h"
#include "criticalguard.h"
#include "kernel.h"

namespace Mark3 {

//---------------------------------------------------------------------------
Coroutine::~Coroutine()
{
    if (m_pclOwner != CoScheduler::GetStopList()) {
        Kernel::Panic(PANIC_ACTIVE_COROUTINE_DESCOPED);
    }

    const auto cs = CriticalGuard{};
    m_pclOwner->Remove(this);
}

//---------------------------------------------------------------------------
void Coroutine::Init(PORT_PRIO_TYPE uPriority_,
CoroutineHandler pfHandler_, void* pvContext_)
{
    m_bQueued = false;
    m_pfHandler = pfHandler_;
    m_pvContext = pvContext_;
    m_uPriority = uPriority_;

    m_pclOwner = CoScheduler::GetStopList();

    const auto cs = CriticalGuard{};
    m_pclOwner->Add(this);
}

//---------------------------------------------------------------------------
void Coroutine::Run()
{
    { // Begin critical section
        const auto cs = CriticalGuard{};
        m_pclOwner->Remove(this);
        m_pclOwner = CoScheduler::GetStopList();
        m_pclOwner->Add(this);
        m_bQueued = false;
    } // end critical section

    m_pfHandler(this, m_pvContext);
}

//---------------------------------------------------------------------------
void Coroutine::Activate()
{
    const auto cs = CriticalGuard{};

    if (m_bQueued) {
        return;
    }

    m_pclOwner->Remove(this);
    m_pclOwner = CoScheduler::GetCoList(
m_uPriority);
    m_pclOwner->Add(this);
    m_bQueued = true;
}

//---------------------------------------------------------------------------
void Coroutine::SetPriority(PORT_PRIO_TYPE uPriority_)
{
    const auto cs = CriticalGuard{};

    m_pclOwner->Remove(this);
    m_uPriority = uPriority_;
    if (m_bQueued) {
```

```
        m_pclOwner = CoScheduler::GetCoList(
    m_uPriority);
    } else {
        m_pclOwner = CoScheduler::GetStopList();
    }
    m_pclOwner->Add(this);
}

//---------------------------------------------------------------------------
PORT_PRIO_TYPE Coroutine::GetPriority()
{
    return m_uPriority;
}
} // namespace Mark3
```

20.37 /home/moslevin/projects/m3-repo/kernel/src/cosched.cpp File Reference

CoRoutine Scheduler implementation.

```
#include "cosched.h"
#include "criticalguard.h"
```

Namespaces

- Mark3

20.37.1 Detailed Description

CoRoutine Scheduler implementation.

Definition in file cosched.cpp.

20.38 cosched.cpp

```
/*===========================================================================
     _____        _____        _____        _____
 ___|    _|__  __|_    |__  __|__   |__  __| __  |__  _____
|    \  /  | ||    \     ||     |     ||  |/ /     ||___   |
|     \/   | ||     \    ||     \     ||  |  \     ||___   |
|__/\__/|__|_||__|\__\  __||__|\__\  __||__|\__\  __||_____|
    |_____|      |_____|      |_____|      |_____|

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

Copyright (c) 2012 - 2019 m0slevin, all rights reserved.
See license.txt for more information
=========================================================================== */
#include "cosched.h"
#include "criticalguard.h"

namespace Mark3 {
CoList CoScheduler::m_aclPriorities[
    PORT_COROUTINE_PRIORITIES];
CoList CoScheduler::m_clStopList;
CoPrioMap CoScheduler::m_clPrioMap;

//---------------------------------------------------------------------------
void CoScheduler::Init()
{
    m_clStopList.Init();
    for (auto i = 0; i < PORT_COROUTINE_PRIORITIES; i++) {
        m_aclPriorities[i].SetPriority(i);
        m_aclPriorities[i].SetPrioMap(&m_clPrioMap);
```

```
    }
}

//---------------------------------------------------------------------------
CoPrioMap* CoScheduler::GetPrioMap()
{
    return &m_clPrioMap;
}

//---------------------------------------------------------------------------
CoList* CoScheduler::GetStopList()
{
    return &m_clStopList;
}

//---------------------------------------------------------------------------
CoList* CoScheduler::GetCoList(PORT_PRIO_TYPE uPriority_)
{
    if (uPriority_ >= PORT_COROUTINE_PRIORITIES) {
        return nullptr;
    }
    return &m_aclPriorities[uPriority_];
}

//---------------------------------------------------------------------------
Coroutine* CoScheduler::Schedule()
{
    const auto cs = CriticalGuard{};

    auto uPriority = m_clPrioMap.HighestPriority();
    if (0 == uPriority) {
        return nullptr;
    }
    uPriority--;
    return m_aclPriorities[uPriority].GetHead();
}
} // namespace Mark3
```

20.39 /home/moslevin/projects/m3-repo/kernel/src/eventflag.cpp File Reference

Event Flag Blocking Object/IPC-Object implementation.

```
#include "mark3.h"
```

20.39.1 Detailed Description

Event Flag Blocking Object/IPC-Object implementation.

Definition in file eventflag.cpp.

20.40 eventflag.cpp

```
/*============================================================================
     _____         _____         _____         _____
 ___|    _|__  __|_    |__  __|__   |__  __| __  |__  ______
|    \  /  | ||    \     ||     |     ||  |/ /     ||___   |
|     \/   | ||     \    ||     \     ||     \     ||___   |
|__/\__/|__|_||__|\__\  __||__|\__\  __||__|\__\  __||______|
    |_____|      |_____|      |_____|      |_____|

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

Copyright (c) 2012 - 2019 m0slevin, all rights reserved.
See license.txt for more information
============================================================================*/
#include "mark3.h"

```

```
#if KERNEL_EVENT_FLAGS

namespace Mark3
{
namespace
{
    //---------------------------------------------------------------------------
    void TimedEventFlag_Callback(Thread* pclOwner_, void* pvData_)
    {
        KERNEL_ASSERT(nullptr != pclOwner_);
        KERNEL_ASSERT(nullptr != pvData_);

        auto* pclEventFlag = static_cast<EventFlag*>(pvData_);

        pclOwner_->SetExpired(true);
        pclOwner_->SetEventFlagMask(0);

        pclEventFlag->WakeMe(pclOwner_);
        if (pclOwner_->GetCurPriority() >= Scheduler::GetCurrentThread()->GetCurPriority()) {
            Thread::Yield();
        }
    }
} // anonymous namespace
//---------------------------------------------------------------------------
EventFlag::~EventFlag()
{
    // If there are any threads waiting on this object when it goes out
    // of scope, set a kernel panic.
    if (nullptr != m_clBlockList.HighestWaiter()) {
        Kernel::Panic(PANIC_ACTIVE_EVENTFLAG_DESCOPED);
    }
}

//---------------------------------------------------------------------------
void EventFlag::Init()
{
    KERNEL_ASSERT(!m_clBlockList.GetHead());
    m_u16SetMask = 0;
    SetInitialized();
}

//---------------------------------------------------------------------------
void EventFlag::WakeMe(Thread* pclChosenOne_)
{
    KERNEL_ASSERT(IsInitialized());
    KERNEL_ASSERT(nullptr != pclChosenOne_);

    UnBlock(pclChosenOne_);
}

//---------------------------------------------------------------------------
uint16_t EventFlag::Wait_i(uint16_t u16Mask_, EventFlagOperation eMode_, uint32_t u32TimeMS_)
{
    KERNEL_ASSERT(eMode_ <= EventFlagOperation::Pending_Unblock);
    KERNEL_ASSERT(IsInitialized());

    auto bThreadYield = false;
    auto bMatch       = false;

    auto clEventTimer = Timer {};
    auto bUseTimer    = false;

    // Ensure we're operating in a critical section while we determine
    // whether or not we need to block the current thread on this object.

    { // Begin critical section
        const auto cs = CriticalGuard{};

        // Check to see whether or not the current mask matches any of the
        // desired bits.
        g_pclCurrent->SetEventFlagMask(u16Mask_);

        if ((EventFlagOperation::All_Set == eMode_) || (EventFlagOperation::All_Clear == eMode_)) {
            // Check to see if the flags in their current state match all of
            // the set flags in the event flag group, with this mask.
            if ((m_u16SetMask & u16Mask_) == u16Mask_) {
                bMatch = true;
                g_pclCurrent->SetEventFlagMask(u16Mask_);

                if (EventFlagOperation::All_Clear == eMode_) {
                    m_u16SetMask &= ~u16Mask_;
                    g_pclCurrent->SetExpired(false);
                }
```

```
            }
        } else if ((EventFlagOperation::Any_Set == eMode_) || (EventFlagOperation::Any_Clear == eMode_)) {
            // Check to see if the existing flags match any of the set flags in
            // the event flag group  with this mask
            if ((m_u16SetMask & u16Mask_) != 0) {
                bMatch = true;
                g_pclCurrent->SetEventFlagMask(m_u16SetMask & u16Mask_);

                if (EventFlagOperation::Any_Clear == eMode_) {
                    m_u16SetMask &= ~u16Mask_;
                    g_pclCurrent->SetExpired(false);
                }
            }
        }

        // We're unable to match this pattern as-is, so we must block.
        if (!bMatch) {
            // Reset the current thread's event flag mask & mode
            g_pclCurrent->SetEventFlagMask(u16Mask_);
            g_pclCurrent->SetEventFlagMode(eMode_);

            if (0u != u32TimeMS_) {
                g_pclCurrent->SetExpired(false);
                clEventTimer.Init();
                clEventTimer.Start(false, u32TimeMS_, TimedEventFlag_Callback, this);
                bUseTimer = true;
            }

            // Add the thread to the object's block-list.
            BlockPriority(g_pclCurrent);

            // Trigger that
            bThreadYield = true;
        }

        // If bThreadYield is set, it means that we've blocked the current thread,
        // and must therefore rerun the scheduler to determine what thread to
        // switch to.
        if (bThreadYield) {
            // Switch threads immediately
            Thread::Yield();
        }
    } // end critical section

    if (bUseTimer && bThreadYield) {
        clEventTimer.Stop();
    }

    return g_pclCurrent->GetEventFlagMask();
}

//---------------------------------------------------------------------------
uint16_t EventFlag::Wait(uint16_t u16Mask_, EventFlagOperation eMode_)
{
    KERNEL_ASSERT(eMode_ <= EventFlagOperation::Pending_Unblock);
    return Wait_i(u16Mask_, eMode_, 0);
}

//---------------------------------------------------------------------------
uint16_t EventFlag::Wait(uint16_t u16Mask_, EventFlagOperation eMode_, uint32_t u32TimeMS_)
{
    KERNEL_ASSERT(eMode_ <= EventFlagOperation::Pending_Unblock);
    return Wait_i(u16Mask_, eMode_, u32TimeMS_);
}

//---------------------------------------------------------------------------
void EventFlag::Set(uint16_t u16Mask_)
{
    KERNEL_ASSERT(IsInitialized());

    auto bReschedule = false;

    const auto cs = CriticalGuard{};
    // Walk through the whole block list, checking to see whether or not
    // the current flag set now matches any/all of the masks and modes of
    // the threads involved.

    m_u16SetMask |= u16Mask_;
    auto u16NewMask = m_u16SetMask;

    // Start at the head of the list, and iterate through until we hit the
    // "head" element in the list again.  Ensure that we handle the case where
```

```
    // we remove the first or last elements in the list, or if there's only
    // one element in the list.

    auto* pclCurrent = m_clBlockList.GetHead();
    // Do nothing when there are no objects blocking.
    if (nullptr != pclCurrent) {
        // First loop - process every thread in the block-list and check to
        // see whether or not the current flags match the event-flag conditions
        // on the thread.
        auto* pclPrev = (Thread*){};
        do {
            pclPrev    = pclCurrent;
            pclCurrent = pclCurrent->GetNext();

            // Read the thread's event mask/mode
            auto u16ThreadMask = pclPrev->GetEventFlagMask();
            auto eThreadMode   = pclPrev->GetEventFlagMode();

            // For the "any" mode - unblock the blocked threads if one or more bits
            // in the thread's bitmask match the object's bitmask
            if ((EventFlagOperation::Any_Set == eThreadMode) || (EventFlagOperation::Any_Clear == eThreadMode)) {
                if ((u16ThreadMask & m_u16SetMask) != 0) {
                    pclPrev->SetEventFlagMode(EventFlagOperation::Pending_Unblock);
                    pclPrev->SetEventFlagMask(m_u16SetMask & u16ThreadMask);
                    bReschedule = true;

                    // If the "clear" variant is set, then clear the bits in the mask
                    // that caused the thread to unblock.
                    if (EventFlagOperation::Any_Clear == eThreadMode) {
                        u16NewMask &= ~(u16ThreadMask & u16Mask_);
                    }
                }
            }
            // For the "all" mode, every set bit in the thread's requested bitmask must
            // match the object's flag mask.
            else if ((EventFlagOperation::All_Set == eThreadMode) || (EventFlagOperation::All_Clear == eThreadMode)) {
                if ((u16ThreadMask & m_u16SetMask) == u16ThreadMask) {
                    pclPrev->SetEventFlagMode(EventFlagOperation::Pending_Unblock);
                    pclPrev->SetEventFlagMask(u16ThreadMask);
                    bReschedule = true;

                    // If the "clear" variant is set, then clear the bits in the mask
                    // that caused the thread to unblock.
                    if (EventFlagOperation::All_Clear == eThreadMode) {
                        u16NewMask &= ~(u16ThreadMask & u16Mask_);
                    }
                }
            }
        }
        // To keep looping, ensure that there's something in the list, and
        // that the next item isn't the head of the list.
        while (pclPrev != m_clBlockList.GetTail());

        // Second loop - go through and unblock all of the threads that
        // were tagged for unblocking.
        pclCurrent   = m_clBlockList.GetHead();
        auto bIsTail = false;
        do {
            pclPrev    = pclCurrent;
            pclCurrent = pclCurrent->GetNext();

            // Check to see if this is the condition to terminate the loop
            if (pclPrev == m_clBlockList.GetTail()) {
                bIsTail = true;
            }

            // If the first pass indicated that this thread should be
            // unblocked, then unblock the thread
            if (pclPrev->GetEventFlagMode() == EventFlagOperation::Pending_Unblock) {
                UnBlock(pclPrev);
            }
        } while (!bIsTail);
    }

    // If we awoke any threads, re-run the scheduler
    if (bReschedule) {
        Thread::Yield();
    }

    // Update the bitmask based on any "clear" operations performed along
    // the way
    m_u16SetMask = u16NewMask;
```

```
}

//---------------------------------------------------------------------------
void EventFlag::Clear(uint16_t u16Mask_)
{
    KERNEL_ASSERT(IsInitialized());

    // Just clear the bitfields in the local object.
    const auto cs = CriticalGuard{};
    m_u16SetMask &= ~u16Mask_;
}

//---------------------------------------------------------------------------
uint16_t EventFlag::GetMask()
{
    KERNEL_ASSERT(IsInitialized());

    // Return the presently held event flag values in this object.  Ensure
    // we get this within a critical section to guarantee atomicity.
    const auto cs = CriticalGuard{};
    auto u16Return = m_u16SetMask;
    return u16Return;
}
} // namespace Mark3
#endif // #if KERNEL_EVENT_FLAGS
```

20.41 /home/moslevin/projects/m3-repo/kernel/src/kernel.cpp File Reference

Kernel initialization and startup code.

```
#include "mark3.h"
```

Namespaces

- Mark3

20.41.1 Detailed Description

Kernel initialization and startup code.

Definition in file kernel.cpp.

20.42 kernel.cpp

```
/*===========================================================================
     _____        _____        _____        _____
 ___|    _|__  __|_    |__  __|__   |__  __| __  |__  ______
|    \  /  | ||    \      ||     |     ||  |/ /     ||___   |
|     \/   | ||     \     ||     \     ||     \     ||___   |
|__/\__/|__|_||__|\__\  __||__|\__\  __||__|\__\  __||______|
    |_____|      |_____|      |_____|      |_____|

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

Copyright (c) 2012 - 2019 m0slevin, all rights reserved.
See license.txt for more information
===========================================================================*/
#include "mark3.h"
namespace Mark3
{
bool      Kernel::m_bIsStarted;
bool      Kernel::m_bIsPanic;
PanicFunc Kernel::m_pfPanic;
```

```

#if KERNEL_THREAD_CREATE_CALLOUT
ThreadCreateCallout Kernel::m_pfThreadCreateCallout;
#endif                                              // #if KERNEL_THREAD_CREATE_CALLOUT
#if KERNEL_THREAD_EXIT_CALLOUT
ThreadExitCallout Kernel::m_pfThreadExitCallout;
#endif                                          // #if KERNEL_THREAD_EXIT_CALLOUT
#if KERNEL_CONTEXT_SWITCH_CALLOUT
ThreadContextCallout Kernel::m_pfThreadContextCallout;
#endif                                                // #if KERNEL_CONTEXT_SWITCH_CALLOUT
DebugPrintFunction Kernel::m_pfDebugPrintFunction;
#if KERNEL_STACK_CHECK
uint16_t Kernel::m_u16GuardThreshold;
#endif // #if KERNEL_STACK_CHECK
uint32_t Kernel::m_u32Ticks;

//---------------------------------------------------------------------------
void Kernel::Init()
{
    // Call port-specific early init function
    ThreadPort::Init();
    AutoAlloc::Init();
    // Initialize the global kernel data - thread-scheduler, and timer-scheduler.
    Scheduler::Init();
    TimerScheduler::Init();
#if KERNEL_STACK_CHECK
    m_u16GuardThreshold = KERNEL_STACK_GUARD_DEFAULT;
#endif // #if KERNEL_STACK_CHECK
}

//---------------------------------------------------------------------------
void Kernel::Start()
{
    ThreadPort::StartThreads();
}

//---------------------------------------------------------------------------
void Kernel::CompleteStart()
{
    m_bIsStarted = true;
}

//---------------------------------------------------------------------------
void Kernel::Panic(uint16_t u16Cause_)
{
    m_bIsPanic = true;
    if (nullptr != m_pfPanic) {
        m_pfPanic(u16Cause_);
    } else {
        while (true) {}
    }
}

//---------------------------------------------------------------------------
void Kernel::DebugPrint(const char* szString_)
{
    KERNEL_ASSERT(nullptr != szString_);
    if (nullptr != m_pfDebugPrintFunction) {
        m_pfDebugPrintFunction(szString_);
    }
}

//---------------------------------------------------------------------------
uint32_t Kernel::GetTicks()
{
    auto cs = CriticalGuard{};
    return m_u32Ticks;
}

} // namespace Mark3
```

20.43 /home/moslevin/projects/m3-repo/kernel/src/ksemaphore.cpp File Reference

Semaphore Blocking-Object Implemenation.

```
#include "mark3.h"
```

Namespaces

- Mark3

20.43.1 Detailed Description

Semaphore Blocking-Object Implemenation.

Definition in file ksemaphore.cpp.

20.44 ksemaphore.cpp

```
/*===========================================================================
     _____        _____        _____        _____
 ___|    _|__  __|_    |__  __|__   |__  __| __  |__  ______
|    \  /  | ||    \      ||     |     ||  |/ /     ||___   |
|     \/   | ||     \     ||     \     ||     \     ||___   |
|__/\__/|__|_||__|\__\  __||__|\__\  __||__|\__\  __||______|
    |_____|      |_____|      |_____|      |_____|

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

Copyright (c) 2012 - 2019 m0slevin, all rights reserved.
See license.txt for more information
===========================================================================*/
#include "mark3.h"

namespace Mark3
{
namespace
{
    //---------------------------------------------------------------------------
    void TimedSemaphore_Callback(Thread* pclOwner_, void* pvData_)
    {
        KERNEL_ASSERT(nullptr != pclOwner_);
        KERNEL_ASSERT(nullptr != pvData_);

        auto* pclSemaphore = static_cast<Semaphore*>(pvData_);

        // Indicate that the semaphore has expired on the thread
        pclOwner_->SetExpired(true);

        // Wake up the thread that was blocked on this semaphore.
        pclSemaphore->WakeMe(pclOwner_);

        if (pclOwner_->GetCurPriority() >= Scheduler::GetCurrentThread()->
GetCurPriority()) {
            Thread::Yield();
        }
    }
} // anonymous namespace

//---------------------------------------------------------------------------
Semaphore::~Semaphore()
{
    // If there are any threads waiting on this object when it goes out
    // of scope, set a kernel panic.
    if (nullptr != m_clBlockList.GetHead()) {
        Kernel::Panic(PANIC_ACTIVE_SEMAPHORE_DESCOPED);
    }
}

//---------------------------------------------------------------------------
void Semaphore::WakeMe(Thread* pclChosenOne_)
{
    KERNEL_ASSERT(pclChosenOne_);
    KERNEL_ASSERT(IsInitialized());

    // Remove from the semaphore waitlist and back to its ready list.
    UnBlock(pclChosenOne_);
}

//---------------------------------------------------------------------------
uint8_t Semaphore::WakeNext()
{
```

```
    auto* pclChosenOne = m_clBlockList.HighestWaiter();
    KERNEL_ASSERT(pclChosenOne);

    // Remove from the semaphore waitlist and back to its ready list.
    UnBlock(pclChosenOne);

    // Call a task switch if higher or equal priority thread
    if (pclChosenOne->GetCurPriority() >= Scheduler::GetCurrentThread()->GetCurPriority()) {
        return 1;
    }
    return 0;
}

//---------------------------------------------------------------------------
void Semaphore::Init(uint16_t u16InitVal_, uint16_t u16MaxVal_)
{
    KERNEL_ASSERT(!m_clBlockList.GetHead());

    // Copy the paramters into the object - set the maximum value for this
    // semaphore to implement either binary or counting semaphores, and set
    // the initial count.  Clear the wait list for this object.
    m_u16Value    = u16InitVal_;
    m_u16MaxValue = u16MaxVal_;

    SetInitialized();
}

//---------------------------------------------------------------------------
bool Semaphore::Post()
{
    KERNEL_ASSERT(IsInitialized());

    auto bThreadWake = false;
    auto bBail       = false;
    // Increment the semaphore count - we can mess with threads so ensure this
    // is in a critical section.  We don't just disable the scheudler since
    // we want to be able to do this from within an interrupt context as well.

    { // Begin critical section
        const auto cs = CriticalGuard{};
        // If nothing is waiting for the semaphore
        if (nullptr == m_clBlockList.GetHead()) {
            // Check so see if we've reached the maximum value in the semaphore
            if (m_u16Value < m_u16MaxValue) {
                // Increment the count value
                m_u16Value++;
            } else {
                // Maximum value has been reached, bail out.
                bBail = true;
            }
        } else {
            // Otherwise, there are threads waiting for the semaphore to be
            // posted, so wake the next one (highest priority goes first).
            bThreadWake = (WakeNext() != 0u);
        }
    } // end critical section

    // If we weren't able to increment the semaphore count, fail out.
    if (bBail) {
        return false;
    }

    // if bThreadWake was set, it means that a higher-priority thread was
    // woken.  Trigger a context switch to ensure that this thread gets
    // to execute next.
    if (bThreadWake) {
        Thread::Yield();
    }
    return true;
}

//---------------------------------------------------------------------------
bool Semaphore::Pend_i(uint32_t u32WaitTimeMS_)
{
    KERNEL_ASSERT(IsInitialized());

    auto clSemTimer = Timer {};
    auto bUseTimer  = false;

    // Once again, messing with thread data - ensure
    // we're doing all of these operations from within a thread-safe context.

    { // Begin critical section
        const auto cs = CriticalGuard{};
        // Check to see if we need to take any action based on the semaphore count
        if (0 != m_u16Value) {
```

```
            // The semaphore count is non-zero, we can just decrement the count
            // and go along our merry way.
            m_u16Value--;
        } else {
            // The semaphore count is zero - we need to block the current thread
            // and wait until the semaphore is posted from elsewhere.
            if (0u != u32WaitTimeMS_) {
                g_pclCurrent->SetExpired(false);
                clSemTimer.Init();
                clSemTimer.Start(false, u32WaitTimeMS_, TimedSemaphore_Callback, this);
                bUseTimer = true;
            }
            BlockPriority(g_pclCurrent);

            // Switch Threads immediately
            Thread::Yield();
        }
    } // End critical section

    if (bUseTimer) {
        clSemTimer.Stop();
        return (g_pclCurrent->GetExpired() == false);
    }
    return true;
}

//---------------------------------------------------------------------------
// Redirect the untimed pend API to the timed pend, with a null timeout.
void Semaphore::Pend()
{
    Pend_i(0);
}

//---------------------------------------------------------------------------
bool Semaphore::Pend(uint32_t u32WaitTimeMS_)
{
    return Pend_i(u32WaitTimeMS_);
}

//---------------------------------------------------------------------------
uint16_t Semaphore::GetCount()
{
    KERNEL_ASSERT(IsInitialized());

    auto cs = CriticalGuard{};
    return m_u16Value;
}
} // namespace Mark3
```

20.45 /home/moslevin/projects/m3-repo/kernel/src/ll.cpp File Reference

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

```
#include "mark3.h"
```

Namespaces

- Mark3

20.45.1 Detailed Description

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

Definition in file ll.cpp.

20.46 ll.cpp

```
/*===========================================================================
     _____        _____        _____        _____
 ___|    _|__  __|_    |__  __|__   |__  __| __  |__  ______
|    \  /  | ||    \      ||     |     ||  |/ /     ||___   |
|     \/   | ||     \     ||     \     ||     \     ||___   |
|__/\__/|__|_||__|\__\  __||__|\__\  __||__|\__\  __||______|
    |_____|      |_____|      |_____|      |_____|

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

Copyright (c) 2012 - 2019 m0slevin, all rights reserved.
See license.txt for more information
===========================================================================*/
#include "mark3.h"

namespace Mark3
{
//---------------------------------------------------------------------------
void LinkListNode::ClearNode()
{
    next = nullptr;
    prev = nullptr;
}

//---------------------------------------------------------------------------
void DoubleLinkList::Add(LinkListNode* node_)
{
    KERNEL_ASSERT(nullptr != node_);

    node_->prev = m_pclTail;
    node_->next = nullptr;

    // If the list is empty, initilize the head
    if (nullptr == m_pclHead) {
        m_pclHead = node_;
    }
    // Otherwise, adjust the tail's next pointer
    else {
        m_pclTail->next = node_;
    }

    // Move the tail node, and assign it to the new node just passed in
    m_pclTail = node_;
}

//---------------------------------------------------------------------------
void DoubleLinkList::Remove(LinkListNode* node_)
{
    KERNEL_ASSERT(nullptr != node_);

    if (nullptr != node_->prev) {
        if (node_->prev->next != node_) {
            Kernel::Panic(PANIC_LIST_UNLINK_FAILED);
        }
        node_->prev->next = node_->next;
    }
    if (nullptr != node_->next) {
        if (node_->next->prev != node_) {
            Kernel::Panic(PANIC_LIST_UNLINK_FAILED);
        }
        node_->next->prev = node_->prev;
    }
    if (node_ == m_pclHead) {
        m_pclHead = node_->next;
    }
    if (node_ == m_pclTail) {
        m_pclTail = node_->prev;
    }
    node_->ClearNode();
}

//---------------------------------------------------------------------------
void CircularLinkList::Add(LinkListNode* node_)
{
    KERNEL_ASSERT(nullptr != node_);

    if (nullptr == m_pclHead) {
        // If the list is empty, initilize the nodes
        m_pclHead = node_;
        m_pclTail = node_;
    } else {
        // Move the tail node, and assign it to the new node just passed in
        m_pclTail->next = node_;
    }
```

```

    // Add a node to the end of the linked list.
    node_->prev = m_pclTail;
    node_->next = m_pclHead;

    m_pclTail       = node_;
    m_pclHead->prev = node_;
}

//---------------------------------------------------------------------------
void CircularLinkList::Remove(LinkListNode* node_)
{
    KERNEL_ASSERT(nullptr != node_);

    // Check to see if this is the head of the list...
    if ((node_ == m_pclHead) && (m_pclHead == m_pclTail)) {
        // Clear the head and tail pointers - nothing else left.
        m_pclHead = nullptr;
        m_pclTail = nullptr;
        return;
    }

    // Verify that all nodes are properly connected
    if ((node_->prev->next != node_) || (node_->next->prev != node_)) {
        Kernel::Panic(PANIC_LIST_UNLINK_FAILED);
    }

    // This is a circularly linked list - no need to check for connection,
    // just remove the node.
    node_->next->prev = node_->prev;
    node_->prev->next = node_->next;

    if (node_ == m_pclHead) {
        m_pclHead = m_pclHead->next;
    }
    if (node_ == m_pclTail) {
        m_pclTail = m_pclTail->prev;
    }
    node_->ClearNode();
}

//---------------------------------------------------------------------------
void CircularLinkList::PivotForward()
{
    if (nullptr != m_pclHead) {
        m_pclHead = m_pclHead->next;
        m_pclTail = m_pclTail->next;
    }
}

//---------------------------------------------------------------------------
void CircularLinkList::PivotBackward()
{
    if (nullptr != m_pclHead) {
        m_pclHead = m_pclHead->prev;
        m_pclTail = m_pclTail->prev;
    }
}

//---------------------------------------------------------------------------
void CircularLinkList::InsertNodeBefore(LinkListNode* node_, LinkListNode* insert_)
{
    KERNEL_ASSERT(nullptr != node_);
    KERNEL_ASSERT(nullptr != insert_);

    node_->next = insert_;
    node_->prev = insert_->prev;

    if (nullptr != insert_->prev) {
        insert_->prev->next = node_;
    }
    insert_->prev = node_;
}
} // namespace Mark3
```

20.47 /home/moslevin/projects/m3-repo/kernel/src/lockguard.cpp File Reference

Mutex RAII helper class.

```
#include "mark3.h"
```

Namespaces

- Mark3

20.47.1 Detailed Description

Mutex RAII helper class.

Definition in file lockguard.cpp.

20.48 lockguard.cpp

```
/*===========================================================================
     _____        _____        _____        _____
 ___|    _|__  __|_    |__  __|__   |__  __| __  |__  ______
|    \  /  | ||    \     ||     |     ||  |/ /     ||___   |
|     \/   | ||     \    ||     \     ||  |    \     ||___   |
|__/\__/|__|_||__|\__\  __||__|\__\  __||__|\__\  __||______|
    |_____|      |_____|      |_____|      |_____|

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

Copyright (c) 2018 m0slevin, all rights reserved.
See license.txt for more information
===========================================================================*/
#include "mark3.h"

namespace Mark3
{
//---------------------------------------------------------------------------
LockGuard::LockGuard(Mutex* pclMutex_)
    : m_bIsAcquired { true }
    , m_pclMutex { pclMutex_ }
{
    KERNEL_ASSERT(nullptr != m_pclMutex);
    m_pclMutex->Claim();
}

//---------------------------------------------------------------------------
LockGuard::LockGuard(Mutex* pclMutex_, uint32_t u32TimeoutMs_)
    : m_pclMutex { pclMutex_ }
{
    KERNEL_ASSERT(nullptr != pclMutex_);
    m_bIsAcquired = m_pclMutex->Claim(u32TimeoutMs_);
}

//---------------------------------------------------------------------------
LockGuard::~LockGuard()
{
    if (m_bIsAcquired) {
        m_pclMutex->Release();
    }
}

} // namespace Mark3
```

20.49 /home/moslevin/projects/m3-repo/kernel/src/mailbox.cpp File Reference

Mailbox + Envelope IPC mechanism.

```
#include "mark3.h"
```

Namespaces

- Mark3

20.49.1 Detailed Description

Mailbox + Envelope IPC mechanism.

Definition in file mailbox.cpp.

20.50 mailbox.cpp

```
/*===========================================================================
     _____     _____     _____     _____
 ___|    _|__ __|_    |__ __|__   |__ __| __  |__  _____
|    \  /  | ||    \    ||     |     ||  |/ /    ||___   |
|     \/   | ||     \   ||     \     ||     \    ||___   |
|__/\__/|__|_||__|\__\ __||__|\__\ __||__|\__\ __||_____|
    |_____|      |_____|     |_____|     |_____|

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

Copyright (c) 2012 - 2019 m0slevin, all rights reserved.
See license.txt for more information
===========================================================================*/
#include "mark3.h"
namespace Mark3
{
//---------------------------------------------------------------------------
Mailbox::~Mailbox()
{
    // If the mailbox isn't empty on destruction, kernel panic.
    if (m_u16Free != m_u16Count) {
        Kernel::Panic(PANIC_ACTIVE_MAILBOX_DESCOPED);
    }
}

//---------------------------------------------------------------------------
void Mailbox::Init(void* pvBuffer_, uint16_t u16BufferSize_, uint16_t u16ElementSize_)
{
    KERNEL_ASSERT(u16BufferSize_);
    KERNEL_ASSERT(u16ElementSize_);
    KERNEL_ASSERT(nullptr != pvBuffer_);

    m_pvBuffer       = pvBuffer_;
    m_u16ElementSize = u16ElementSize_;

    m_u16Count = (u16BufferSize_ / u16ElementSize_);
    m_u16Free  = m_u16Count;

    m_u16Head = 0;
    m_u16Tail = 0;

    // We use the counting semaphore to implement blocking - with one element
    // in the mailbox corresponding to a post/pend operation in the semaphore.
    m_clRecvSem.Init(0, m_u16Free);

    // Binary semaphore is used to track any threads that are blocked on a
    // "send" due to lack of free slots.
    m_clSendSem.Init(0, 1);
}

//---------------------------------------------------------------------------
Mailbox* Mailbox::Init(uint16_t u16BufferSize_, uint16_t u16ElementSize_)
{
    KERNEL_ASSERT(u16BufferSize_);
    KERNEL_ASSERT(u16ElementSize_);

    auto* pclNew   = AutoAlloc::NewObject<Mailbox, AutoAllocType::MailBox>();
    auto* pvBuffer = AutoAlloc::NewRawData(u16BufferSize_);

    KERNEL_ASSERT(nullptr != pclNew);
    KERNEL_ASSERT(nullptr != pvBuffer);

    if (!pclNew) {
        return nullptr;
    }
    if (!pvBuffer) {
        AutoAlloc::DestroyObject<Mailbox, AutoAllocType::MailBox>(pclNew);
        return nullptr;
    }

    pclNew->Init(pvBuffer, u16BufferSize_, u16ElementSize_);
```

```
    return pclNew;
}

//---------------------------------------------------------------------------
void Mailbox::Receive(void* pvData_)
{
    KERNEL_ASSERT(nullptr != pvData_);
    Receive_i(pvData_, false, 0);
}

//---------------------------------------------------------------------------
bool Mailbox::Receive(void* pvData_, uint32_t u32TimeoutMS_)
{
    KERNEL_ASSERT(nullptr != pvData_);
    return Receive_i(pvData_, false, u32TimeoutMS_);
}

//---------------------------------------------------------------------------
void Mailbox::ReceiveTail(void* pvData_)
{
    KERNEL_ASSERT(nullptr != pvData_);
    Receive_i(pvData_, true, 0);
}

//---------------------------------------------------------------------------
bool Mailbox::ReceiveTail(void* pvData_, uint32_t u32TimeoutMS_)
{
    KERNEL_ASSERT(nullptr != pvData_);
    return Receive_i(pvData_, true, u32TimeoutMS_);
}

//---------------------------------------------------------------------------
bool Mailbox::Send(void* pvData_)
{
    KERNEL_ASSERT(nullptr != pvData_);
    return Send_i(pvData_, false, 0);
}

//---------------------------------------------------------------------------
bool Mailbox::SendTail(void* pvData_)
{
    KERNEL_ASSERT(nullptr != pvData_);
    return Send_i(pvData_, true, 0);
}

//---------------------------------------------------------------------------
bool Mailbox::Send(void* pvData_, uint32_t u32TimeoutMS_)
{
    KERNEL_ASSERT(nullptr != pvData_);
    return Send_i(pvData_, false, u32TimeoutMS_);
}

//---------------------------------------------------------------------------
bool Mailbox::SendTail(void* pvData_, uint32_t u32TimeoutMS_)
{
    KERNEL_ASSERT(nullptr != pvData_);
    return Send_i(pvData_, true, u32TimeoutMS_);
}

//---------------------------------------------------------------------------
bool Mailbox::Send_i(const void* pvData_, bool bTail_, uint32_t u32TimeoutMS_)
{
    KERNEL_ASSERT(nullptr != pvData_);

    void* pvDst = nullptr;

    auto bRet        = false;
    auto bSchedState = Scheduler::SetScheduler(false);
    auto bBlock      = false;
    auto bDone       = false;

    while (!bDone) {
        // Try to claim a slot first before resorting to blocking.
        if (bBlock) {
            bDone = true;
            Scheduler::SetScheduler(bSchedState);
            m_clSendSem.Pend(u32TimeoutMS_);
            Scheduler::SetScheduler(false);
        }

        { // Begin critical section
            const auto cs = CriticalGuard{};
            // Ensure we have a free slot before we attempt to write data
            if (0u != m_u16Free) {
                m_u16Free--;

                if (bTail_) {
```

```
                    pvDst = GetTailPointer();
                    MoveTailBackward();
                } else {
                    MoveHeadForward();
                    pvDst = GetHeadPointer();
                }
                bRet  = true;
                bDone = true;
            } else if (0u != u32TimeoutMS_) {
                bBlock = true;
            } else {
                bDone = true;
            }
        } // End critical section
    }

    // Copy data to the claimed slot, and post the counting semaphore
    if (bRet) {
        CopyData(pvData_, pvDst, m_u16ElementSize);
    }

    Scheduler::SetScheduler(bSchedState);

    if (bRet) {
        m_clRecvSem.Post();
    }

    return bRet;
}

//---------------------------------------------------------------------------
bool Mailbox::Receive_i(void* pvData_, bool bTail_, uint32_t u32WaitTimeMS_)
{
    KERNEL_ASSERT(nullptr != pvData_);
    auto* pvSrc = (const void*){};

    if (!m_clRecvSem.Pend(u32WaitTimeMS_)) {
        // Failed to get the notification from the counting semaphore in the
        // time allotted.  Bail.
        return false;
    }

    // Disable the scheduler while we do this -- this ensures we don't have
    // multiple concurrent readers off the same queue, which could be problematic
    // if multiple writes occur during reads, etc.
    auto bSchedState = Scheduler::SetScheduler(false);

    // Update the head/tail indexes, and get the associated data pointer for
    // the read operation.

    { // Begin critical section
        const auto cs = CriticalGuard{};
        m_u16Free++;
        if (bTail_) {
            MoveTailForward();
            pvSrc = GetTailPointer();
        } else {
            pvSrc = GetHeadPointer();
            MoveHeadBackward();
        }
    } // end critical section

    KERNEL_ASSERT(pvSrc);
    CopyData(pvSrc, pvData_, m_u16ElementSize);

    Scheduler::SetScheduler(bSchedState);

    // Unblock a thread waiting for a free slot to send to
    m_clSendSem.Post();

    return true;
}
} // namespace Mark3
```

20.51 /home/moslevin/projects/m3-repo/kernel/src/message.cpp File Reference

Inter-thread communications via message passing.

```
#include "mark3.h"
```

Namespaces

- Mark3

20.51.1 Detailed Description

Inter-thread communications via message passing.

Definition in file message.cpp.

20.52 message.cpp

```
/*===========================================================================
     _____     _____     _____     _____
 ___|    _|__ __|_    |__ __|__   |__ __| __  |__  ______
|    \  /  | ||    \    ||     |     ||  |/ /    ||___   |
|     \/   | ||     \   ||     \     ||     \    ||___   |
|__/\__/|__|_||__|\__\  __||__|\__\  __||__|\__\ __||______|
    |_____|      |_____|     |_____|     |_____|

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

Copyright (c) 2012 - 2019 m0slevin, all rights reserved.
See license.txt for more information
===========================================================================*/
#include "mark3.h"
namespace Mark3
{
//---------------------------------------------------------------------------
void MessagePool::Init()
{
    m_clList.Init();
}

//---------------------------------------------------------------------------
void MessagePool::Push(Message* pclMessage_)
{
    KERNEL_ASSERT(pclMessage_);

    const auto cs = CriticalGuard{};
    m_clList.Add(pclMessage_);
}

//---------------------------------------------------------------------------
Message* MessagePool::Pop()
{
    const auto cs = CriticalGuard{};
    auto* pclRet = m_clList.GetHead();
    if (nullptr != pclRet) {
        m_clList.Remove(pclRet);
    }
    return pclRet;
}

//------------------------------------------------------------------------
Message* MessagePool::GetHead()
{
    return m_clList.GetHead();
}

//---------------------------------------------------------------------------
void MessageQueue::Init()
{
    m_clSemaphore.Init(0, 255);
}

//---------------------------------------------------------------------------
Message* MessageQueue::Receive()
{
    return Receive_i(0);
}

//---------------------------------------------------------------------------
Message* MessageQueue::Receive(uint32_t u32TimeWaitMS_)
{
```

```
    return Receive_i(u32TimeWaitMS_);
}

//---------------------------------------------------------------------------
Message* MessageQueue::Receive_i(uint32_t u32TimeWaitMS_)
{
    // Block the current thread on the counting semaphore
    if (!m_clSemaphore.Pend(u32TimeWaitMS_)) {
        return nullptr;
    }

    const auto cs = CriticalGuard{};
    // Pop the head of the message queue and return it
    auto* pclRet = m_clLinkList.GetHead();
    m_clLinkList.Remove(pclRet);

    return pclRet;
}

//---------------------------------------------------------------------------
void MessageQueue::Send(Message* pclSrc_)
{
    KERNEL_ASSERT(pclSrc_);

    CriticalSection::Enter();

    // Add the message to the head of the linked list
    m_clLinkList.Add(pclSrc_);

    CriticalSection::Exit();

    // Post the semaphore, waking the blocking thread for the queue.
    m_clSemaphore.Post();
}

//---------------------------------------------------------------------------
uint16_t MessageQueue::GetCount()
{
    return m_clSemaphore.GetCount();
}
} // namespace Mark3
```

20.53 /home/moslevin/projects/m3-repo/kernel/src/mutex.cpp File Reference

Mutual-exclusion object.

```
#include "mark3.h"
```

Namespaces

- Mark3

20.53.1 Detailed Description

Mutual-exclusion object.

Definition in file mutex.cpp.

20.54 mutex.cpp

```
/*===========================================================================
     _____     _____     _____     _____
 ___|    _|__ __|_    |__ __|__   |__ __| __  |__  ______
|    \  /  | ||    \    ||     |     ||  |/ /     ||___   |
|     \/   | ||     \   ||     \     ||     \     ||___   |
|__/\__/|__|_||__|\__\ __||__|\__\ __||__|\__\  __||_______|
    |_____|      |_____|      |_____|      |_____|

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

Copyright (c) 2012 - 2019 m0slevin, all rights reserved.
See license.txt for more information
===========================================================================*/
#include "mark3.h"
namespace Mark3
{
namespace
{
    //---------------------------------------------------------------------------
    void TimedMutex_Callback(Thread* pclOwner_, void* pvData_)
    {
        KERNEL_ASSERT(nullptr != pclOwner_);
        KERNEL_ASSERT(nullptr != pvData_);

        auto* pclMutex = static_cast<Mutex*>(pvData_);

        // Indicate that the semaphore has expired on the thread
        pclOwner_->SetExpired(true);

        // Wake up the thread that was blocked on this semaphore.
        pclMutex->WakeMe(pclOwner_);

        if (pclOwner_->GetCurPriority() >= Scheduler::GetCurrentThread()->
GetCurPriority()) {
            Thread::Yield();
        }
    }
} // anonymous namespace

//---------------------------------------------------------------------------
Mutex::~Mutex()
{
    // If there are any threads waiting on this object when it goes out
    // of scope, set a kernel panic.
    if (nullptr != m_clBlockList.GetHead()) {
        Kernel::Panic(PANIC_ACTIVE_MUTEX_DESCOPED);
    }
}

//---------------------------------------------------------------------------
void Mutex::WakeMe(Thread* pclOwner_)
{
    KERNEL_ASSERT(nullptr != pclOwner_);
    // Remove from the semaphore waitlist and back to its ready list.
    UnBlock(pclOwner_);
}

//---------------------------------------------------------------------------
uint8_t Mutex::WakeNext()
{
    // Get the highest priority waiter thread
    auto* pclChosenOne = m_clBlockList.HighestWaiter();
    KERNEL_ASSERT(pclChosenOne);

    // Unblock the thread
    UnBlock(pclChosenOne);

    // The chosen one now owns the mutex
    m_pclOwner = pclChosenOne;

    // Signal a context switch if it's a greater than or equal to the current priority
    if (pclChosenOne->GetCurPriority() >= Scheduler::GetCurrentThread()->
GetCurPriority()) {
        return 1;
    }
    return 0;
}

//---------------------------------------------------------------------------
void Mutex::Init(bool bRecursive_)
{
    // Cannot re-init a mutex which has threads blocked on it
    KERNEL_ASSERT(!m_clBlockList.GetHead());

```

```
    // Reset the data in the mutex
    m_bReady      = true;    // The mutex is free.
    m_uMaxPri     = 0;       // Set the maximum priority inheritence state
    m_pclOwner    = nullptr; // Clear the mutex owner
    m_u8Recurse   = 0;       // Reset recurse count
    m_bRecursive = bRecursive_;
    SetInitialized();
}

//---------------------------------------------------------------------------
bool Mutex::Claim_i(uint32_t u32WaitTimeMS_)
{
    KERNEL_ASSERT(IsInitialized());

    auto clTimer   = Timer {};
    auto bUseTimer = false;

    // Disable the scheduler while claiming the mutex - we're dealing with all
    // sorts of private thread data, can't have a thread switch while messing
    // with internal data structures.
    Scheduler::SetScheduler(false);

    // Check to see if the mutex is claimed or not
    if (false != m_bReady) {
        // Mutex isn't claimed, claim it.
        m_bReady     = false;
        m_u8Recurse = 0;
        m_uMaxPri    = g_pclCurrent->GetPriority();
        m_pclOwner   = g_pclCurrent;

        Scheduler::SetScheduler(true);
        return true;
    }

    // If the mutex is already claimed, check to see if this is the owner thread,
    // since we allow the mutex to be claimed recursively.
    if (m_bRecursive && (g_pclCurrent == m_pclOwner)) {
        // Ensure that we haven't exceeded the maximum recursive-lock count
        KERNEL_ASSERT((m_u8Recurse < 255));
        m_u8Recurse++;

        // Increment the lock count and bail
        Scheduler::SetScheduler(true);
        return true;
    }

    // The mutex is claimed already - we have to block now.  Move the
    // current thread to the list of threads waiting on the mutex.
    if (0u != u32WaitTimeMS_) {
        g_pclCurrent->SetExpired(false);
        clTimer.Init();
        clTimer.Start(false, u32WaitTimeMS_, TimedMutex_Callback, this);
        bUseTimer = true;
    }
    BlockPriority(g_pclCurrent);

    // Check if priority inheritence is necessary.  We do this in order
    // to ensure that we don't end up with priority inversions in case
    // multiple threads are waiting on the same resource.
    if (m_uMaxPri <= g_pclCurrent->GetPriority()) {
        m_uMaxPri = g_pclCurrent->GetPriority();

        auto* pclTemp = m_clBlockList.GetHead();
        while (nullptr != pclTemp) {
            pclTemp->InheritPriority(m_uMaxPri);
            if (m_clBlockList.GetTail() == pclTemp) {
                break;
            }
            pclTemp = pclTemp->GetNext();
        }
        m_pclOwner->InheritPriority(m_uMaxPri);
    }

    // Done with thread data -reenable the scheduler
    Scheduler::SetScheduler(true);

    // Switch threads if this thread acquired the mutex
    Thread::Yield();

    if (bUseTimer) {
        clTimer.Stop();
        return (false == g_pclCurrent->GetExpired());
    }
    return true;
}

//---------------------------------------------------------------------------
```

```
void Mutex::Claim(void)
{
    Claim_i(0);
}

//---------------------------------------------------------------------------
bool Mutex::Claim(uint32_t u32WaitTimeMS_)
{
    return Claim_i(u32WaitTimeMS_);
}

//---------------------------------------------------------------------------
void Mutex::Release()
{
    KERNEL_ASSERT(IsInitialized());

    auto bSchedule = false;

    // Disable the scheduler while we deal with internal data structures.
    Scheduler::SetScheduler(false);

    // This thread had better be the one that owns the mutex currently...
    KERNEL_ASSERT((g_pclCurrent == m_pclOwner));

    // If the owner had claimed the lock multiple times, decrease the lock
    // count and return immediately.
    if (m_bRecursive && (0u != m_u8Recurse)) {
        m_u8Recurse--;
        Scheduler::SetScheduler(true);
        return;
    }

    // Restore the thread's original priority
    if (g_pclCurrent->GetCurPriority() != g_pclCurrent->
GetPriority()) {
        g_pclCurrent->SetPriority(g_pclCurrent->
GetPriority());

        // In this case, we want to reschedule
        bSchedule = true;
    }

    // No threads are waiting on this semaphore?
    if (nullptr == m_clBlockList.GetHead()) {
        // Re-initialize the mutex to its default values
        m_bReady    = true;
        m_uMaxPri   = 0;
        m_pclOwner = nullptr;
    } else {
        // Wake the highest priority Thread pending on the mutex
        if (0u != WakeNext()) {
            // Switch threads if it's higher or equal priority than the current thread
            bSchedule = true;
        }
    }

    // Must enable the scheduler again in order to switch threads.
    Scheduler::SetScheduler(true);
    if (bSchedule) {
        // Switch threads if a higher-priority thread was woken
        Thread::Yield();
    }
}
} // namespace Mark3
```

20.55 /home/moslevin/projects/m3-repo/kernel/src/notify.cpp File Reference

Lightweight thread notification - blocking object.

```
#include "mark3.h"
```

Namespaces

- Mark3

20.55.1 Detailed Description

Lightweight thread notification - blocking object.

Definition in file notify.cpp.

20.56 notify.cpp

```
/*===========================================================================
     _____     _____     _____     _____
 ___|    _|__ __|_    |__ __|__   |__ __| __  |__  ______
|    \  /  | ||    \     ||     |     ||  |/ /     ||___   |
|     \/   | ||     \    ||     \     ||  \      ||___   |
|__/\__/|__|_||__|\__\  __||__|\__\  __||__|\__\  __||______|
    |_____|      |_____|      |_____|      |_____|

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

Copyright (c) 2012 - 2019 m0slevin, all rights reserved.
See license.txt for more information
===========================================================================*/
#include "mark3.h"
namespace Mark3
{
namespace
{
    //---------------------------------------------------------------------------
    void TimedNotify_Callback(Thread* pclOwner_, void* pvData_)
    {
        KERNEL_ASSERT(nullptr != pclOwner_);
        KERNEL_ASSERT(nullptr != pvData_);

        auto* pclNotify = static_cast<Notify*>(pvData_);

        // Indicate that the semaphore has expired on the thread
        pclOwner_->SetExpired(true);

        // Wake up the thread that was blocked on this semaphore.
        pclNotify->WakeMe(pclOwner_);

        if (pclOwner_->GetCurPriority() >= Scheduler::GetCurrentThread()->
GetCurPriority()) {
            Thread::Yield();
        }
    }
} // anonymous namespace

//---------------------------------------------------------------------------
Notify::~Notify()
{
    // If there are any threads waiting on this object when it goes out
    // of scope, set a kernel panic.
    if (nullptr != m_clBlockList.GetHead()) {
        Kernel::Panic(PANIC_ACTIVE_NOTIFY_DESCOPED);
    }
}

//---------------------------------------------------------------------------
void Notify::Init(void)
{
    KERNEL_ASSERT(!m_clBlockList.GetHead());
    SetInitialized();

    m_bPending = false;
}

//---------------------------------------------------------------------------
void Notify::Signal(void)
{
    KERNEL_ASSERT(IsInitialized());

    auto bReschedule = false;

    { // Begin critical section
        const auto cs = CriticalGuard{};
        auto* pclCurrent = m_clBlockList.GetHead();
        if (nullptr == pclCurrent) {
            m_bPending = true;
        } else {
```

```
            while (nullptr != pclCurrent) {
                UnBlock(pclCurrent);
                if (!bReschedule && (pclCurrent->GetCurPriority() >=
Scheduler::GetCurrentThread()->GetCurPriority())) {
                    bReschedule = true;
                }
                pclCurrent = m_clBlockList.GetHead();
            }
            m_bPending = false;
        }
    } // end critical section

    if (bReschedule) {
        Thread::Yield();
    }
}

//---------------------------------------------------------------------------
void Notify::Wait(bool* pbFlag_)
{
    KERNEL_ASSERT(nullptr != pbFlag_);
    KERNEL_ASSERT(IsInitialized());

    auto bEarlyExit = false;
    { // Begin critical section
        const auto cs = CriticalGuard{};
        if (!m_bPending) {
            Block(g_pclCurrent);
            if (nullptr != pbFlag_) {
                *pbFlag_ = false;
            }
        } else {
            m_bPending = false;
            bEarlyExit = true;
        }
    } // End critical section

    if (bEarlyExit) {
        return;
    }

    Thread::Yield();
    if (nullptr != pbFlag_) {
        *pbFlag_ = true;
    }
}

//---------------------------------------------------------------------------
bool Notify::Wait(uint32_t u32WaitTimeMS_, bool* pbFlag_)
{
    KERNEL_ASSERT(nullptr != pbFlag_);
    KERNEL_ASSERT(IsInitialized());

    auto bUseTimer     = false;
    auto bEarlyExit    = false;
    auto clNotifyTimer = Timer {};

    { // Begin critical section
        const auto cs = CriticalGuard{};
        if (!m_bPending) {
            if (0u != u32WaitTimeMS_) {
                bUseTimer = true;
                g_pclCurrent->SetExpired(false);

                clNotifyTimer.Init();
                clNotifyTimer.Start(false, u32WaitTimeMS_, TimedNotify_Callback, this);
            }

            Block(g_pclCurrent);

            if (nullptr != pbFlag_) {
                *pbFlag_ = false;
            }
        } else {
            m_bPending = false;
            bEarlyExit = true;
        }
    } // end critical section

    if (bEarlyExit) {
        return true;
    }

    Thread::Yield();

    if (bUseTimer) {
        clNotifyTimer.Stop();
```

```
        return (g_pclCurrent->GetExpired() == false);
    }

    if (nullptr != pbFlag_) {
        *pbFlag_ = true;
    }

    return true;
}

//---------------------------------------------------------------------------
void Notify::WakeMe(Thread* pclChosenOne_)
{
    KERNEL_ASSERT(nullptr != pclChosenOne_);
    KERNEL_ASSERT(IsInitialized());

    UnBlock(pclChosenOne_);
}
} // namespace Mark3
```

20.57 /home/moslevin/projects/m3-repo/kernel/src/profile.cpp File Reference

Code profiling utilities.

```
#include "mark3.h"
```

Namespaces

- Mark3

20.57.1 Detailed Description

Code profiling utilities.

Definition in file profile.cpp.

20.58 profile.cpp

```
/*===========================================================================
     _____        _____        _____        _____
 ___|    _|__  __|_    |__  __|__   |__  __| __  |__  ______
|    \  /  | ||    \      ||     |     ||  |/ /     ||___   |
|     \/   | ||     \     ||     \     ||  |\ \     ||___   |
|__/\__/|__|_||__|\__\  __||__|\__\  __||__|\__\  __||______|
    |_____|      |_____|      |_____|      |_____|

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

Copyright (c) 2012 - 2019 m0slevin, all rights reserved.
See license.txt for more information
===========================================================================*/
#include "mark3.h"
namespace Mark3
{
//---------------------------------------------------------------------------
void ProfileTimer::Init()
{
    m_u32StartTicks = 0;
    m_u32Cumulative = 0;
    m_u16Iterations = 0;
    m_bActive       = false;
}

```

```
//---------------------------------------------------------------------------
void ProfileTimer::Start()
{
    if (!m_bActive) {
        { // Begin critical section
            const auto cs = CriticalGuard{};
            m_u32StartTicks = Kernel::GetTicks();
        } // End Critical Section
        m_bActive = true;
    }
}

//---------------------------------------------------------------------------
void ProfileTimer::Stop()
{
    if (m_bActive) {
        uint32_t u32Final;
        { // Begin critical section
            const auto cs = CriticalGuard{};
            u32Final = Kernel::GetTicks();
            // Compute total for current iteration...
            m_u32CurrentIteration = u32Final -
m_u32StartTicks;
            m_u32Cumulative += m_u32CurrentIteration;
            m_u16Iterations++;
        } // End critical section
        m_bActive = false;
    }
}

//---------------------------------------------------------------------------
uint32_t ProfileTimer::GetAverage()
{
    if (0u != m_u16Iterations) {
        return (m_u32Cumulative + static_cast<uint32_t>(
m_u16Iterations / 2)) / static_cast<uint32_t>(m_u16Iterations);
    }
    return 0;
}

//---------------------------------------------------------------------------
uint32_t ProfileTimer::GetCurrent()
{
    if (m_bActive) {
        uint32_t u32Current;
        { // Begin critical section
            const auto cs = CriticalGuard{};
            u32Current = Kernel::GetTicks() - m_u32StartTicks;
        } // End critical section
        return u32Current;
    }
    return m_u32CurrentIteration;
}
} // namespace Mark3
```

20.59 /home/moslevin/projects/m3-repo/kernel/src/public/atomic.h File Reference

Basic Atomic Operations.

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

Namespaces

- Mark3
- Mark3::Atomic

  The Atomic namespace This utility module provides primatives for atomic operations - that is, operations that are guaranteed to execute uninterrupted. Basic atomic primatives provided here include Set/Add/Subtract, as well as an atomic test-and-set.

Functions

- template<typename T >
  T Mark3::Atomic::Set (T *pSource_, T val_)

  Set Set a variable to a given value in an uninterruptable operation.
- template<typename T >
  T Mark3::Atomic::Add (T *pSource_, T val_)

  Add Add a value to a variable in an uninterruptable operation.
- template<typename T >
  T Mark3::Atomic::Sub (T *pSource_, T val_)

  Sub Subtract a value from a variable in an uninterruptable operation.
- bool Mark3::Atomic::TestAndSet (bool *pbLock)

  TestAndSet Test to see if a variable is set, and set it if is not already set. This is an uninterruptable operation.

20.59.1 Detailed Description

Basic Atomic Operations.

Definition in file atomic.h.

20.60 atomic.h

```
/*===============================================================================
     _____        _____        _____        _____
 ___|    _|__  __|_    |__  __|__   |__  __| __  |__  ______
|    \  /  | ||    \     ||    |     ||  |/ /     ||___   |
|     \/   | ||     \    ||    \     ||     \    ||___   |
|__/\__/|__|_||__|\__\  __||__|\__\  __||__|\__\  __||_____|
    |_____|      |_____|      |_____|      |_____|

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

Copyright (c) 2012 - 2019 m0slevin, all rights reserved.
See license.txt for more information
===============================================================================*/
#pragma once

#include "kerneltypes.h"
#include "mark3cfg.h"
#include "ithreadport.h"
#include "kerneldebug.h"

namespace Mark3
{
namespace Atomic
{
    template <typename T> T Set(T* pSource_, T val_)
    {
        KERNEL_ASSERT(nullptr != pSource_);

        const auto cs = CriticalGuard{};
        auto ret        = *pSource_;
        *pSource_ = val_;
        return ret;
    }

    template <typename T> T Add(T* pSource_, T val_)
    {
        KERNEL_ASSERT(nullptr != pSource_);

        const auto cs = CriticalGuard{};
        auto ret = *pSource_;
        *pSource_ += val_;
        return ret;
    }

    template <typename T> T Sub(T* pSource_, T val_)
    {
```

```
        KERNEL_ASSERT(nullptr != pSource_);

        const auto cs = CriticalGuard{};
        auto ret = *pSource_;
        *pSource_ -= val_;
        return ret;
    }

    bool TestAndSet(bool* pbLock);
} // namespace Atomic
} // namespace Mark3
```

20.61 /home/moslevin/projects/m3-repo/kernel/src/public/autoalloc.h File Reference

Automatic memory allocation for kernel objects.

```
#include <stddef.h>
#include <stdint.h>
#include <stdbool.h>
#include "mark3cfg.h"
```

Classes

- class Mark3::AutoAlloc

  The AutoAlloc class. This class provides an object-allocation interface for both kernel objects and user-defined types. This class supplies callouts for alloc/free that use object-type metadata to determine how objects may be allocated, allowing a user to create custom dynamic memory implementations for specific object types and sizes. As a result, the user-defined allocators can avoid the kinds of memory fragmentation and exhaustion issues that occur in typical embedded systems in which a single heap is used to satisfy all allocations in the application.

Namespaces

- Mark3

Typedefs

- using Mark3::AutoAllocAllocator_t = void ∗(∗)(AutoAllocType eType_, size_t sSize_)
- using Mark3::AutoAllocFree_t = void(∗)(AutoAllocType eType_, void ∗pvObj_)

Enumerations

- enum Mark3::AutoAllocType : uint8_t {
  Mark3::AutoAllocType::EventFlag, Mark3::AutoAllocType::MailBox, Mark3::AutoAllocType::Message, Mark3::AutoAllocType::MessagePool,
  Mark3::AutoAllocType::MessageQueue, Mark3::AutoAllocType::Mutex, Mark3::AutoAllocType::Notify, Mark3::AutoAllocType::Semaphore,
  Mark3::AutoAllocType::Thread, Mark3::AutoAllocType::Timer, Mark3::AutoAllocType::ConditionVariable, Mark3::AutoAllocType::ReaderWriterLock,
  Mark3::AutoAllocType::User, Mark3::AutoAllocType::Raw = 0xFF }

20.61.1 Detailed Description

Automatic memory allocation for kernel objects.

Definition in file autoalloc.h.

20.62 autoalloc.h

```
/*===========================================================================
     _____        _____        _____        _____
 ___|    _|__  __|_    |__  __|__   |__  __| __  |__  ______
|    \  /  | ||    \      ||     |     ||  |/ /     ||___   |
|     \/   | ||     \     ||     \     ||     \     ||___   |
|__/\__/|__|_||__|\__\  __||__|\__\  __||__|\__\  __||______|
    |_____|      |_____|      |_____|      |_____|

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

Copyright (c) 2012 - 2019 m0slevin, all rights reserved.
See license.txt for more information
=========================================================================== */
#pragma once

#include <stddef.h>
#include <stdint.h>
#include <stdbool.h>
#include "mark3cfg.h"

namespace Mark3
{
//---------------------------------------------------------------------------
// Define function pointer types used for interfacing with an external heap.
//---------------------------------------------------------------------------
enum class AutoAllocType : uint8_t {
    //-- Kernel object types
    EventFlag,
    MailBox,
    Message,
    MessagePool,
    MessageQueue,
    Mutex,
    Notify,
    Semaphore,
    Thread,
    Timer,
    ConditionVariable,
    ReaderWriterLock,
    //-- Allow for users to define their own object types beginning with AutoAllocType_t::User
    User,
    //--
    Raw = 0xFF
};

//---------------------------------------------------------------------------
using AutoAllocAllocator_t = void* (*)(AutoAllocType eType_, size_t sSize_
);
using AutoAllocFree_t      = void (*)(AutoAllocType eType_, void* pvObj_);

//---------------------------------------------------------------------------
// Forward declaration of kernel objects that can be auotomatically allocated.
class EventFlag;
class Mailbox;
class Message;
class MessagePool;
class MessageQueue;
class Mutex;
class Notify;
class Semaphore;
class Thread;
class Timer;
class ReaderWriterLock;
class ConditionVariable;

class AutoAlloc
{
public:
    static void Init(void);

    static void SetAllocatorFunctions(AutoAllocAllocator_t pfAllocator_,
```

```
AutoAllocFree_t pfFree_);

    template <typename T, AutoAllocType e> static T* NewObject()
    {
        auto* pvObj = Allocate(e, sizeof(T));
        if (pvObj) {
            return new (pvObj) T();
        }
        return 0;
    }

    template <typename T, AutoAllocType e> static void DestroyObject(T* pObj_)
    {
        pObj_->~T();
        Free(e, pObj_);
    }

    static void* NewUserTypeAllocation(uint8_t eUserType_);

    static void DestroyUserTypeAllocation(uint8_t eUserType_, void* pvObj_);

    static void* NewRawData(size_t sSize_);

    static void DestroyRawData(void* pvData_);

private:
    static void* Allocate(AutoAllocType eType_, size_t sSize_);
    static void  Free(AutoAllocType eType_, void* pvObj_);

    static AutoAllocAllocator_t m_pfAllocator;
    static AutoAllocFree_t      m_pfFree;
};
} // namespace Mark3
```

20.63 /home/moslevin/projects/m3-repo/kernel/src/public/blocking.h File Reference

Blocking object base class declarations.

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

Classes

- class Mark3::BlockingObject

  The BlockingObject class. 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.

Namespaces

- Mark3

20.63.1 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.

20.64 blocking.h

```
/*===========================================================================
     _____        _____        _____        _____
 ___|    _|__  __|_    |__  __|__   |__  __| __  |__  ______
|    \  /  | ||    \      ||     |     ||  |/ /     ||___   |
|     \/   | ||     \     ||     \     ||     \     ||___   |
|__/\__/|__|_||__|\__\  __||__|\__\  __||__|\__\  __||______|
    |_____|      |_____|      |_____|      |_____|

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

Copyright (c) 2012 - 2019 m0slevin, all rights reserved.
See license.txt for more information
=========================================================================== */
#pragma once

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

#include "ll.h"
#include "threadlist.h"

namespace Mark3
{
class Thread;

//---------------------------------------------------------------------------
class BlockingObject
{
public:
    BlockingObject() { m_u8Initialized =
m_uBlockingInvalidCookie; }
    ~BlockingObject() { m_u8Initialized =
m_uBlockingInvalidCookie; }

protected:
    void Block(Thread* pclThread_);

    void BlockPriority(Thread* pclThread_);

    void UnBlock(Thread* pclThread_);
```

```
    void SetInitialized(void) { m_u8Initialized =
m_uBlockingInitCookie; }

    bool IsInitialized(void) { return (m_u8Initialized ==
m_uBlockingInitCookie); }

    // Cookies used to determine whether or not an object has been initialized
    static constexpr auto m_uBlockingInvalidCookie = uint8_t { 0x3C };
    static constexpr auto m_uBlockingInitCookie    = uint8_t { 0xC3 };

    ThreadList m_clBlockList;

    uint8_t m_u8Initialized;
};
} // namespace Mark3
```

20.65 /home/moslevin/projects/m3-repo/kernel/src/public/colist.h File Reference

CoRoutine List structure implementation.

```
#include "mark3cfg.h"
#include "coroutine.h"
```

Classes

- class Mark3::CoList

  The CoList class The CoList class implements a circular-linked-listed structure for coroutine objects. The intent of this object is to maintain a list of active coroutine objects with a specific priority or state, to ensure that a freshly-schedulable co-routine always exists at the head of the list.

Namespaces

- Mark3

20.65.1 Detailed Description

CoRoutine List structure implementation.

Definition in file colist.h.

20.66 colist.h

```
/*===========================================================================
     _____     _____     _____     _____
 ___|    _|__ __|_    |__ __|__   |__ __| __  |__  _____
|    \  /  | ||    \    ||     |    ||  |/ /    ||___   |
|     \/   | ||     \   ||     \    ||     \    ||___   |
|__/\__/|__|_||__|\__\ __||__|\__\ __||__|\__\ __||_____|
    |_____|      |_____|      |_____|      |_____|

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

Copyright (c) 2012 - 2019 m0slevin, all rights reserved.
See license.txt for more information
=========================================================================== */
#pragma once
```

```

#include "mark3cfg.h"
#include "coroutine.h"

namespace Mark3 {
class Coroutine;

class CoList : public TypedCircularLinkList<Coroutine>
{
public:

    void SetPrioMap(CoPrioMap* pclPrioMap_);

    void SetPriority(PORT_PRIO_TYPE uPriority_);

    void Add(Coroutine* pclCoroutine_);

    void Remove(Coroutine* pclCoroutine_);

private:
    CoPrioMap* m_pclPrioMap;
    uint8_t m_uPriority;
};
} // namespace Mark3
```

20.67 /home/moslevin/projects/m3-repo/kernel/src/public/condvar.h File Reference

Condition Variable implementation.

```
#include "mark3cfg.h"
#include "ksemaphore.h"
#include "mutex.h"
#include <stddef.h>
```

Classes

- class Mark3::ConditionVariable

  The ConditionVariable class This class implements a condition variable. This is a synchronization object that allows multiple threads to block, each waiting for specific signals unique to them. Access to the specified condition is guarded by a mutex that is supplied by the caller. This object can permit multiple waiters that can be unblocked one-at-a-time via signalling, or unblocked all at once via broadcasting. This object is built upon lower-level primatives, and is somewhat more heavyweight than the primative types supplied by the kernel.

Namespaces

- Mark3

20.67.1 Detailed Description

Condition Variable implementation.

Definition in file condvar.h.

20.68 condvar.h

```
/*===========================================================================
     _____        _____        _____        _____
 ___|    _|__  __|_    |__  __|__   |__  __| __  |__  ______
|    \  /  | ||    \      ||     |     ||  |/ /     ||___   |
|     \/   | ||     \     ||     \     ||     \     ||___   |
|__/\__/|__|_||__|\__\  __||__|\__\  __||__|\__\  __||______|
    |_____|      |_____|      |_____|      |_____|

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

Copyright (c) 2012 - 2019 m0slevin, all rights reserved.
See license.txt for more information
===========================================================================*/
#pragma once

#include "mark3cfg.h"
#include "ksemaphore.h"
#include "mutex.h"

#include <stddef.h>

namespace Mark3
{
class ConditionVariable
{
public:
    void* operator new(size_t sz, void* pv) { return reinterpret_cast<
    ConditionVariable*>(pv); }

    void Init();

    void Wait(Mutex* pclMutex_);

    bool Wait(Mutex* pclMutex_, uint32_t u32WaitTimeMS_);

    void Signal();

    void Broadcast();

private:
    Mutex     m_clMutex;
    Semaphore m_clSemaphore;
    uint8_t   m_u8Waiters;
};
} // namespace Mark3
```

20.69 /home/moslevin/projects/m3-repo/kernel/src/public/coroutine.h File Reference

CoRoutine implementation.

```
#include "mark3cfg.h"
#include "ll.h"
#include "priomapl1.h"
#include "priomapl2.h"
```

Classes

- class Mark3::Coroutine

  The Coroutine class implements a lightweight, run-to-completion task that forms the basis for co-operative task scheduling in Mark3. Coroutines are designed to be run from a singular context, and scheduled as a result of events occurring from threads, timers, interrupt sources, or other co-routines.

Namespaces

- Mark3

Typedefs

- using Mark3::CoPrioMap = PriorityMapL1< PORT_PRIO_TYPE, PORT_COROUTINE_PRIORITIES >
- using Mark3::CoroutineHandler = void(∗)(Coroutine ∗pclCaller_, void ∗pvContext_)

20.69.1 Detailed Description

CoRoutine implementation.

Definition in file coroutine.h.

20.70 coroutine.h

```
/*===========================================================================
     _____        _____        _____        _____
 ___|    _|__  __|_    |__  __|__   |__  __| __  |__  ______
|    \  /  | ||    \      ||     |     ||  |/ /     ||___   |
|     \/   | ||     \     ||     \     ||     \     ||___   |
|__/\__/|__|_||__|\__\  __||__|\__\  __||__|\__\  __||______|
    |_____|      |_____|      |_____|      |_____|

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

Copyright (c) 2012 - 2019 m0slevin, all rights reserved.
See license.txt for more information
=========================================================================== */
#pragma once

#include "mark3cfg.h"
#include "ll.h"
#include "priomapl1.h"
#include "priomapl2.h"

namespace Mark3 {

// Priority map type declaration, based on port configuration
#if PORT_COROUTINE_PRIORITIES <= (PORT_PRIO_MAP_WORD_SIZE * 8u)
using CoPrioMap =
PriorityMapL1<PORT_PRIO_TYPE, PORT_COROUTINE_PRIORITIES>
;
#else
using CoPrioMap =
PriorityMapL2<PORT_PRIO_TYPE, PORT_COROUTINE_PRIORITIES>
;
#endif

// Forward declarations
class CoList;
class Coroutine;

// CoRoutine functino handler type definition
using CoroutineHandler = void (*)(Coroutine* pclCaller_, void* pvContext_);

class Coroutine : public TypedLinkListNode<Coroutine>
{
public:

    ~Coroutine();

    void Init(PORT_PRIO_TYPE uPriority_, CoroutineHandler pfHandler_,
void* pvContext_);

    void Run();

    void Activate();

    void SetPriority(PORT_PRIO_TYPE uPriority_);

    PORT_PRIO_TYPE GetPriority();

private:
    CoList* m_pclOwner;
    CoroutineHandler m_pfHandler;
    void* m_pvContext;
    PORT_PRIO_TYPE m_uPriority;
    bool m_bQueued;
};
} // namespace Mark3
```

20.71 /home/moslevin/projects/m3-repo/kernel/src/public/cosched.h File Reference

CoRoutine Scheduler implementation.

```
#include "mark3cfg.h"
#include "coroutine.h"
#include "colist.h"
```

Classes

- class Mark3::CoScheduler

  The CoScheduler class. This class implements the coroutine scheduler. Similar to the Mark3 thread scheduler, the highest-priority active object is scheduled / returned for execution. If no active co-routines are available to be scheduled, then the scheduler returns nullptr.

Namespaces

- Mark3

20.71.1 Detailed Description

CoRoutine Scheduler implementation.

Definition in file cosched.h.

20.72 cosched.h

```
/*===========================================================================
     _____        _____        _____        _____
 ___|    _|__  __|_    |__  __|__   |__  __| __  |__  ______
|    \  /  | ||    \      ||     |     ||  |/ /     ||___   |
|     \/   | ||     \     ||     \     ||     \     ||___   |
|__/\__/|__|_||__|\__\  __||__|\__\  __||__|\__\  __||______|
    |_____|      |_____|      |_____|      |_____|

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

Copyright (c) 2012 - 2019 m0slevin, all rights reserved.
See license.txt for more information
=========================================================================== */
#pragma once

#include "mark3cfg.h"
#include "coroutine.h"
#include "colist.h"

namespace Mark3 {
class CoScheduler
{
public:

    static void Init();

    static CoPrioMap* GetPrioMap();

    static CoList* GetStopList();

    static CoList* GetCoList(PORT_PRIO_TYPE uPriority_);

    static Coroutine* Schedule();

private:
    static CoList m_aclPriorities[PORT_COROUTINE_PRIORITIES];
    static CoList m_clStopList;
    static CoPrioMap m_clPrioMap;
};
} // namespace Mark3
```

20.73 /home/moslevin/projects/m3-repo/kernel/src/public/criticalguard.h File Reference

RAII Critical Section Implementation.

```
#include "mark3cfg.h"
#include "criticalsection.h"
```

Classes

- class Mark3::CriticalGuard

  The CriticalGuard class. This class provides an implemention of RAII for critical sections. Object creation results in a critical section being invoked. The subsequent destructor call results in the critical section being released.

Namespaces

- Mark3

20.73.1 Detailed Description

RAII Critical Section Implementation.

Definition in file criticalguard.h.

20.74 criticalguard.h

```
/*===========================================================================
     _____       _____        _____       _____
 ___|    _|__  __|_    |__  __|__   |__  __| __  |__  ______
|    \  /  | ||    \      ||     |     ||  |/ /     ||___   |
|     \/   | ||     \     ||     \     ||     \     ||___   |
|__/\__/|__|_||__|\__\  __||__|\__\  __||__|\__\  __||______|
    |_____|      |_____|      |_____|      |_____|

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

Copyright (c) 2012 - 2019 m0slevin, all rights reserved.
See license.txt for more information
===========================================================================*/
#pragma once

#include "mark3cfg.h"
#include "criticalsection.h"

namespace Mark3 {

//---------------------------------------------------------------------------
class CriticalGuard {
public:
    CriticalGuard() {
        CriticalSection::Enter();
    }

    ~CriticalGuard() {
        CriticalSection::Exit();
    }
};
} // namespace Mark3
```

20.75 /home/moslevin/projects/m3-repo/kernel/src/public/criticalsection.h File Reference

Critical Section Support.

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

Classes

- class Mark3::CriticalSection

  The CriticalSection class. This class implements a portable CriticalSection interface based on macros/inline functions that are implemented as part of each port.

Namespaces

- Mark3

20.75.1 Detailed Description

Critical Section Support.

Definition in file criticalsection.h.

20.76 criticalsection.h

```
/*===========================================================================
     _____        _____        _____        _____
 ___|    _|__  __|_    |__  __|__   |__  __| __  |__  ______
|    \  /  | ||    \      ||     |     ||  |/ /    ||___   |
|     \/   | ||     \     ||     \     ||     \    ||___   |
|__/\__/|__|_||__|\__\  __||__|\__\  __||__|\__\  __||______|
    |_____|      |_____|      |_____|      |_____|

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

Copyright (c) 2012 - 2019 m0slevin, all rights reserved.
See license.txt for more information
===========================================================================*/
#pragma once

#include "mark3cfg.h"
#include "threadport.h"

namespace Mark3 {

//---------------------------------------------------------------------------
class CriticalSection {
public:

    static inline void Enter() {
        PORT_CS_ENTER();
    }

    static inline void Exit() {
        PORT_CS_EXIT();
    }

    static inline K_WORD NestingCount() {
        return PORT_CS_NESTING();
    }
};

} // namespace Mark3
```

20.77 /home/moslevin/projects/m3-repo/kernel/src/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 Mark3::EventFlag

  The EventFlag class. This class implements 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.

Namespaces

- Mark3

20.77.1 Detailed Description

Event Flag Blocking Object/IPC-Object definition.

Definition in file eventflag.h.

20.78 eventflag.h

```
/*===========================================================================
     _____        _____        _____        _____
 ___|    _|__  __|_    |__  __|__   |__  __| __  |__  ______
|    \  /  | ||    \      ||     |     ||  |/ /     ||___   |
|     \/   | ||     \     ||     \     ||     \     ||___   |
|__/\__/|__|_||__|\__\  __||__|\__\  __||__|\__\  __||______|
    |_____|      |_____|      |_____|      |_____|

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

Copyright (c) 2012 - 2019 m0slevin, all rights reserved.
See license.txt for more information
===========================================================================*/
#pragma once

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

#if KERNEL_EVENT_FLAGS
namespace Mark3
{
//---------------------------------------------------------------------------
class EventFlag : public BlockingObject
```

```
{
public:
    void* operator new(size_t sz, void* pv) { return reinterpret_cast<EventFlag*>(pv); };
    ~EventFlag();

    void Init();

    uint16_t Wait(uint16_t u16Mask_, EventFlagOperation eMode_);

    uint16_t Wait(uint16_t u16Mask_, EventFlagOperation eMode_, uint32_t u32TimeMS_);

    void WakeMe(Thread* pclChosenOne_);

    void Set(uint16_t u16Mask_);

    void Clear(uint16_t u16Mask_);

    uint16_t GetMask();

private:
    uint16_t Wait_i(uint16_t u16Mask_, EventFlagOperation eMode_, uint32_t
u32TimeMS_);

    uint16_t m_u16SetMask;
};
} // namespace Mark3
#endif // #if KERNEL_EVENT_FLAGS
```

20.79 /home/moslevin/projects/m3-repo/kernel/src/public/ithreadport.h File Reference

Thread porting interface.

```
#include <stdint.h>
```

Classes

- class Mark3::ThreadPort

  The ThreadPort Class defines the target-specific functions required by the kernel for threading.

Namespaces

- Mark3

20.79.1 Detailed Description

Thread porting interface.

Definition in file ithreadport.h.

20.80 ithreadport.h

```
/*===========================================================================
     _____        _____        _____        _____
 ___|    _|__  __|_    |__  __|__   |__  __| __  |__  ______
|    \  /  | ||    \      ||     |     ||  |/ /     ||___   |
|     \/   | ||     \     ||     \     ||     \     ||___   |
|__/\__/|__|_||__|\__\  __||__|\__\  __||__|\__\  __||______|
    |_____|      |_____|      |_____|      |_____|

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

Copyright (c) 2012 - 2019 m0slevin, all rights reserved.
See license.txt for more information
===========================================================================*/
#pragma once

#include <stdint.h>

namespace Mark3
{
//---------------------------------------------------------------------------
class Thread;
class ThreadPort
{
public:
    static void Init() {}

    static void StartThreads();
    friend class Thread;

private:
    static void InitStack(Thread* pstThread_);
};
} // namespace Mark3
```

20.81 /home/moslevin/projects/m3-repo/kernel/src/public/kernel.h File Reference

Kernel initialization and startup class.

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

Classes

- class Mark3::Kernel

    The Kernel Class encapsulates all of the kernel startup, configuration and management functions.

Namespaces

- Mark3

Typedefs

- using DebugPrintFunction = void(∗)(const char ∗szString_)

20.81.1 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.

20.81.2 Typedef Documentation

20.81.2.1 DebugPrintFunction

```
using DebugPrintFunction = void (*)(const char* szString_)
```

Definition at line 39 of file kernel.h.

20.82 kernel.h

```
00001 /*==========================================================================
00002      _____     _____     _____     _____
00003  ___|    _|__ __|_    |__ __|__   |__ __| __  |__  _____
00004 |    \  /  | ||    \     ||     |     ||  |/ /     ||___   |
00005 |     \/   | ||     \    ||     \     ||  |  \     ||___   |
00006 |__/\__/|__|_||__|\__\  __||__|\__\  __||__|\__\  __||_______|
00007     |_____|      |_____|      |_____|      |_____|
00008
00009 --[Mark3 Realtime Platform]-------------------------------------------------
00010
00011 Copyright (c) 2012 - 2019 m0slevin, all rights reserved.
00012 See license.txt for more information
00013 ==========================================================================*/
00032 #pragma once
00033
00034 #include "mark3cfg.h"
00035 #include "kerneltypes.h"
00036 #include "paniccodes.h"
00037 #include "thread.h"
00038
00039 using DebugPrintFunction = void (*)(const char* szString_);
00040
00041 namespace Mark3
00042 {
00043 //--------------------------------------------------------------------------
00048 class Kernel
00049 {
00050 public:
00058     static void Init();
00059
00071     static void Start();
00072
00078     static void CompleteStart();
00079
00086     static bool IsStarted() { return m_bIsStarted; }
00094     static void SetPanic(PanicFunc pfPanic_) { m_pfPanic = pfPanic_; }
00099     static bool IsPanic() { return m_bIsPanic; }
00104     static void Panic(uint16_t u16Cause_);
00105
```

```
#if KERNEL_THREAD_CREATE_CALLOUT

    static void SetThreadCreateCallout(ThreadCreateCallout pfCreate_) { m_pfThreadCreateCallout = pfCreate_; }
#endif // #if KERNEL_THREAD_CREATE_HOOK

#if KERNEL_THREAD_EXIT_CALLOUT

    static void SetThreadExitCallout(ThreadExitCallout pfExit_) { m_pfThreadExitCallout = pfExit_; }
#endif // #if KERNEL_THREAD_EXIT_CALLOUT

#if KERNEL_CONTEXT_SWITCH_CALLOUT

    static void SetThreadContextSwitchCallout(ThreadContextCallout pfContext_)
    {
        m_pfThreadContextCallout = pfContext_;
    }
#endif // KERNEL_CONTEXT_SWITCH_CALLOUT

    static void SetDebugPrintFunction(DebugPrintFunction pfPrintFunction_)
    {
        m_pfDebugPrintFunction = pfPrintFunction_;
    }

    static void DebugPrint(const char* szString_);

#if KERNEL_THREAD_CREATE_CALLOUT

    static ThreadCreateCallout GetThreadCreateCallout() { return m_pfThreadCreateCallout; }
#endif // #if KERNEL_THREAD_CREATE_HOOK
#if KERNEL_THREAD_EXIT_CALLOUT

    static ThreadExitCallout GetThreadExitCallout() { return m_pfThreadExitCallout; }
#endif // #if KERNEL_THREAD_EXIT_HOOK
#if KERNEL_CONTEXT_SWITCH_CALLOUT

    static ThreadContextCallout GetThreadContextSwitchCallout() { return m_pfThreadContextCallout; }
#endif // #if KERNEL_CONTEXT_SWITCH_CALLOUT
#if KERNEL_STACK_CHECK
    static void     SetStackGuardThreshold(uint16_t u16Threshold_) { m_u16GuardThreshold = u16Threshold_; }
    static uint16_t GetStackGuardThreshold() { return m_u16GuardThreshold; }
#endif // #if KERNEL_STACK_CHECK

    static void     Tick() { m_u32Ticks++; }
    static uint32_t GetTicks();

private:
    static bool      m_bIsStarted;
    static bool      m_bIsPanic;
    static PanicFunc m_pfPanic;

#if KERNEL_THREAD_CREATE_CALLOUT
    static ThreadCreateCallout m_pfThreadCreateCallout;
#endif                                                    // #if KERNEL_THREAD_CREATE_HOOK
#if KERNEL_THREAD_EXIT_CALLOUT
    static ThreadExitCallout m_pfThreadExitCallout;
#endif                                                // #if KERNEL_THREAD_EXIT_HOOK
#if KERNEL_CONTEXT_SWITCH_CALLOUT
    static ThreadContextCallout m_pfThreadContextCallout;
#endif                                                      // #if KERNEL_CONTEXT_SWITCH_CALLOUT
    static DebugPrintFunction m_pfDebugPrintFunction;
#if KERNEL_STACK_CHECK
    static uint16_t m_u16GuardThreshold;
#endif // #if KERNEL_STACK_CHECK
    static uint32_t m_u32Ticks;
};

} // namespace Mark3
```

20.83 /home/moslevin/projects/m3-repo/kernel/src/public/kerneldebug.h File Reference

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

```
#include "mark3cfg.h"
#include "paniccodes.h"
#include "kernel.h"
```

Namespaces

- Mark3

Macros

- #define KERNEL_ASSERT(x)

20.83.1 Detailed Description

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

Definition in file kerneldebug.h.

20.83.2 Macro Definition Documentation

20.83.2.1 KERNEL_ASSERT

```
#define KERNEL_ASSERT(
                x )
```

Definition at line 36 of file kerneldebug.h.

20.84 kerneldebug.h

```
00001 /*===========================================================================
00002      _____     _____     _____     _____
00003  ___|    _|__ __|_    |__ __|__   |__ __| __  |__  ______
00004 |    \  /  | ||    \     ||     |     ||  |/ /     ||___   |
00005 |     \/   | ||     \    ||     \     ||     \     ||___   |
00006 |__/\__/|__|_||__|\__\  __||__|\__\  __||__|\__\  __||______|
00007     |_____|      |_____|      |_____|      |_____|
00008
00009 --[Mark3 Realtime Platform]--------------------------------------------------
00010
00011 Copyright (c) 2012 - 2019 m0slevin, all rights reserved.
00012 See license.txt for more information
00013 ===========================================================================*/
00019 #pragma once
00020
00021 #include "mark3cfg.h"
00022 #include "paniccodes.h"
00023 #include "kernel.h"
00024
00025 //---------------------------------------------------------------------------
00026 namespace Mark3
00027 {
00028 #if KERNEL_DEBUG
```

```
#define KERNEL_ASSERT(x)                                                       \
    do {                                                                       \
        if ((x) == 0) {                                                        \
            Kernel::Panic(PANIC_ASSERT_FAILED);                                \
        }                                                                      \
    } while (0);
#else
#define KERNEL_ASSERT(x)
#endif

} // namespace Mark3
```

20.85 /home/moslevin/projects/m3-repo/kernel/src/public/kernelswi.h File Reference

Kernel Software interrupt declarations.

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

Classes

- class Mark3::KernelSWI

  The KernelSWI Class provides the software-interrupt used to implement the context-switching interrupt used by the kernel. This interface must be implemented by target-specific code in the porting layer.

Namespaces

- Mark3

20.85.1 Detailed Description

Kernel Software interrupt declarations.

Definition in file kernelswi.h.

20.86 kernelswi.h

```
/*===========================================================================
     _____     _____     _____     _____
 ___|    _|__ __|_    |__ __|__   |__ __| __  |__  ______
|    \  /  | ||    \    ||     |     ||  |/ /     ||___   |
|     \/   | ||     \   ||     \     ||  |  \     ||___   |
|__/\__/|__|_||__|\__\  __||__|\__\  __||__|\__\ __||______|
    |_____|      |_____|      |_____|      |_____|

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

Copyright (c) 2012 - 2019 m0slevin, all rights reserved.
See license.txt for more information
=========================================================================== */
#pragma once
#include "kerneltypes.h"

//---------------------------------------------------------------------------
```

```
namespace Mark3
{
class KernelSWI
{
public:
    static void Config(void);

    static void Start(void);

    static void Trigger(void);
};
} // namespace Mark3
```

20.87 /home/moslevin/projects/m3-repo/kernel/src/public/kerneltimer.h File Reference

Kernel Timer Class declaration.

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

Classes

- class Mark3::KernelTimer

  The KernelTimer class provides a timer interface used by all time-based scheduling/timer subsystems in the kernel. This interface must be implemented by target-specific code in the porting layer.

Namespaces

- Mark3

20.87.1 Detailed Description

Kernel Timer Class declaration.

Definition in file kerneltimer.h.

20.88 kerneltimer.h

```
/*===========================================================================
     _____        _____        _____        _____
 ___|    _|__  __|_    |__  __|__   |__  __| __  |__  ______
|    \  /  | ||    \      ||     |     ||  |/ /     ||___   |
|     \/   | ||     \     ||     \     ||     \     ||___   |
|__/\__/|__|_||__|\__\  __||__|\__\  __||__|\__\  __||______|
    |_____|      |_____|      |_____|      |_____|

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

Copyright (c) 2012 - 2019 m0slevin, all rights reserved.
See license.txt for more information
=========================================================================== */
#pragma once

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

namespace Mark3
{
//---------------------------------------------------------------------------
class KernelTimer
{
public:
    static void Config(void);

    static void Start(void);

    static void Stop(void);
};
} // namespace Mark3
```

20.89 /home/moslevin/projects/m3-repo/kernel/src/public/kerneltypes.h File Reference

Basic data type primatives used throughout the OS.

```
#include <stdint.h>
#include <stdbool.h>
#include <stddef.h>
```

Namespaces

- Mark3

Typedefs

- using Mark3::PanicFunc = void(*)(uint16_t u16PanicCode_)
- using Mark3::IdleFunc = void(*)()
- using Mark3::ThreadEntryFunc = void(*)(void *pvArg_)

Enumerations

- enum Mark3::EventFlagOperation : uint8_t {
  Mark3::EventFlagOperation::All_Set = 0, Mark3::EventFlagOperation::Any_Set, Mark3::EventFlag↩ Operation::All_Clear, Mark3::EventFlagOperation::Any_Clear,
  Mark3::EventFlagOperation::Pending_Unblock }
- enum Mark3::ThreadState : uint8_t {
  Mark3::ThreadState::Exit = 0, Mark3::ThreadState::Ready, Mark3::ThreadState::Blocked, Mark3::Thread↩ State::Stop,
  Mark3::ThreadState::Invalid }

20.89.1 Detailed Description

Basic data type primatives used throughout the OS.

Definition in file kerneltypes.h.

20.90 kerneltypes.h

```
/*===========================================================================
     _____        _____        _____        _____
 ___|    _|__  __|_    |__  __|__   |__  __| __  |__  ______
|    \  /  | ||    \      ||     |     ||  |/ /     ||___   |
|     \/   | ||     \     ||     \     ||     \     ||___   |
|__/\__/|__|_||__|\__\  __||__|\__\  __||__|\__\  __||______|
    |_____|      |_____|      |_____|      |_____|

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

Copyright (c) 2012 - 2019 m0slevin, all rights reserved.
See license.txt for more information
===========================================================================*/
#include <stdint.h>
#include <stdbool.h>
#include <stddef.h>
```

```

#pragma once
namespace Mark3
{
//---------------------------------------------------------------------------
using PanicFunc = void (*)(uint16_t u16PanicCode_);

//---------------------------------------------------------------------------
using IdleFunc = void (*)();

//---------------------------------------------------------------------------
using ThreadEntryFunc = void (*)(void* pvArg_);

//---------------------------------------------------------------------------
enum class EventFlagOperation : uint8_t {
    All_Set = 0,
    Any_Set,
    All_Clear,
    Any_Clear,
    Pending_Unblock
};

//---------------------------------------------------------------------------
enum class ThreadState : uint8_t { Exit = 0, Ready, Blocked, Stop, Invalid };

} // namespace Mark3
```

20.91 /home/moslevin/projects/m3-repo/kernel/src/public/ksemaphore.h File Reference

Semaphore Blocking Object class declarations.

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

Classes

- class Mark3::Semaphore

    the Semaphore class provides Binary & Counting semaphore objects, based on BlockingObject base class.

Namespaces

- Mark3

20.91.1 Detailed Description

Semaphore Blocking Object class declarations.

Definition in file ksemaphore.h.

20.92 ksemaphore.h

```
/*=======================================================================
     _____     _____     _____     _____
 ___|    _|__ __|_    |__ __|__   |__ __| __  |__  _____
|    \  /  | ||    \    ||     |     ||  |/ /    ||___   |
|     \/   | ||     \   ||     \     ||  |\ \    ||___   |
|__/\__/|__|_||__|\__\ __||__|\__\ __||__|\__\ __||______|
    |_____|      |_____|     |_____|     |_____|

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

Copyright (c) 2012 - 2019 m0slevin, all rights reserved.
See license.txt for more information
=========================================================================== */
#pragma once

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

#include "blocking.h"
#include "threadlist.h"

namespace Mark3
{
//---------------------------------------------------------------------------
class Semaphore : public BlockingObject
{
public:
    void* operator new(size_t sz, void* pv) { return reinterpret_cast<Semaphore*>(pv); };
    ~Semaphore();

    void Init(uint16_t u16InitVal_, uint16_t u16MaxVal_);

    bool Post();

    void Pend();

    uint16_t GetCount();

    bool Pend(uint32_t u32WaitTimeMS_);

    void WakeMe(Thread* pclChosenOne_);

private:
    uint8_t WakeNext();

    bool Pend_i(uint32_t u32WaitTimeMS_);

    uint16_t m_u16Value;
    uint16_t m_u16MaxValue;
};
} // namespace Mark3
```

20.93 /home/moslevin/projects/m3-repo/kernel/src/public/ll.h File Reference

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.

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

Classes

- class Mark3::LinkListNode

  The LinkListNode Class Basic linked-list node data structure. This data is managed by the linked-list class types, and can be used transparently between them.

- class Mark3::TypedLinkListNode< T >

The [TypedLinkListNode] class The [TypedLinkListNode] class provides a linked-list node type for a specified object type. This can be used with typed link-list data structures to manage lists of objects without having to static-cast between the base type and the derived class.

- class Mark3::LinkList

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

- class Mark3::DoubleLinkList

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

- class Mark3::CircularLinkList

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

- class Mark3::TypedDoubleLinkList< T >

  The TypedDoubleLinkList Class Doubly-linked-list data type, inherited from the base LinkList type, and templated for use with linked-list-node derived data-types.

- class Mark3::TypedCircularLinkList< T >

  The TypedCircularLinkList Class Circular-linked-list data type, inherited from the base LinkList type, and templated for use with linked-list-node derived data-types.

Namespaces

- Mark3

20.93.1 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.

20.94 ll.h

```
/*===========================================================================
     _____        _____        _____        _____
 ___|    _|__  __|_    |__  __|__   |__  __| __  |__  ______
|    \  /  | ||    \      ||     |     ||  |/ /     ||___   |
|     \/   | ||     \     ||     \     ||     \     ||___   |
|__/\__/|__|_||__|\__\  __||__|\__\  __||__|\__\  __||______|
    |_____|      |_____|      |_____|      |_____|

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

Copyright (c) 2012 - 2019 m0slevin, all rights reserved.
See license.txt for more information
=========================================================================== */
#pragma once
#include "kerneltypes.h"

namespace Mark3
```

```
{
//---------------------------------------------------------------------------
class LinkList;
class DoubleLinkList;
class CircularLinkList;

//---------------------------------------------------------------------------
class LinkListNode
{
protected:
    LinkListNode* next;
    LinkListNode* prev;

    LinkListNode() {}

    void ClearNode();

public:
    LinkListNode* GetNext(void) { return next; }
    LinkListNode* GetPrev(void) { return prev; }
    friend class LinkList;
    friend class DoubleLinkList;
    friend class CircularLinkList;
};

//---------------------------------------------------------------------------
template <typename T>
class TypedLinkListNode : public LinkListNode
{
public:
    T* GetNext() { return static_cast<T*>(LinkListNode::GetNext()); }
    T* GetPrev() { return static_cast<T*>(LinkListNode::GetPrev()); }
};
//---------------------------------------------------------------------------
class LinkList
{
protected:
    LinkListNode* m_pclHead;
    LinkListNode* m_pclTail;

public:
    void Init()
    {
        m_pclHead = nullptr;
        m_pclTail = nullptr;
    }

    LinkListNode* GetHead() { return m_pclHead; }

    void SetHead(LinkListNode* pclNode_) { m_pclHead = pclNode_; }

    LinkListNode* GetTail() { return m_pclTail; }

    void SetTail(LinkListNode* pclNode_) { m_pclTail = pclNode_; }
};

//---------------------------------------------------------------------------
class DoubleLinkList : public LinkList
{
public:
    void* operator new(size_t sz, void* pv) { return reinterpret_cast<DoubleLinkList*>(pv); };
    DoubleLinkList()
    {
        m_pclHead = nullptr;
        m_pclTail = nullptr;
    }

    void Add(LinkListNode* node_);

    void Remove(LinkListNode* node_);
};

//---------------------------------------------------------------------------
class CircularLinkList : public LinkList
{
public:
    void* operator new(size_t sz, void* pv) { return (CircularLinkList*)pv; };
    CircularLinkList()
    {
        m_pclHead = nullptr;
        m_pclTail = nullptr;
    }

    void Add(LinkListNode* node_);

    void Remove(LinkListNode* node_);
```

```

    void PivotForward();

    void PivotBackward();

    void InsertNodeBefore(LinkListNode* node_, LinkListNode* insert_);
};

//---------------------------------------------------------------------------
template <typename T>
class TypedDoubleLinkList : public DoubleLinkList
{
public:
    void* operator new(size_t sz, void* pv) { return reinterpret_cast<
TypedDoubleLinkList<T>*>(pv); }

    TypedDoubleLinkList<T>() {
        DoubleLinkList::Init();
    }

    T* GetHead() { return static_cast<T*>(DoubleLinkList::GetHead()); }

    void SetHead(T* pclNode_) { DoubleLinkList::SetHead(pclNode_); }

    T* GetTail() { return static_cast<T*>(DoubleLinkList::GetTail()); }

    void SetTail(T* pclNode_) { DoubleLinkList::SetTail(pclNode_); }

    void Add(T* pNode_)
    {
        DoubleLinkList::Add(pNode_);
    }

    void Remove(T* pNode_)
    {
        DoubleLinkList::Remove(pNode_);
    }
};

//---------------------------------------------------------------------------
template <typename T>
class TypedCircularLinkList : public CircularLinkList
{
public:
    void* operator new(size_t sz, void* pv) { return reinterpret_cast<
TypedCircularLinkList<T>*>(pv); }

    TypedCircularLinkList<T>() {
        CircularLinkList::Init();
    }

    T* GetHead() { return static_cast<T*>(CircularLinkList::GetHead()); }

    void SetHead(T* pclNode_) { CircularLinkList::SetHead(pclNode_); }

    T* GetTail() { return static_cast<T*>(CircularLinkList::GetTail()); }

    void SetTail(T* pclNode_) { CircularLinkList::SetTail(pclNode_); }

    void Add(T* pNode_)
    {
        CircularLinkList::Add(pNode_);
    }

    void Remove(T* pNode_)
    {
        CircularLinkList::Remove(pNode_);
    }

    void InsertNodeBefore(T* pNode_, T* pInsert_)
    {
        CircularLinkList::InsertNodeBefore(pNode_, pInsert_);
    }
};
} // namespace Mark3
```

20.95 /home/moslevin/projects/m3-repo/kernel/src/public/lockguard.h File Reference

Mutex RAII helper class.

```
#include "mark3.h"
```

Classes

- class Mark3::LockGuard

  The LockGuard class. This class provides RAII locks based on Mark3's kernel Mutex object. Note that Mark3 does not support exceptions, so care must be taken to ensure that this object is only used where that constraint can be met.

Namespaces

- Mark3

20.95.1 Detailed Description

Mutex RAII helper class.

Definition in file lockguard.h.

20.96 lockguard.h

```
/*===========================================================================
     _____        _____        _____        _____
 ___|    _|__  __|_    |__  __|__   |__  __| __  |__  ______
|    \  /  | ||    \      ||     |     ||  |/ /    ||___   |
|     \/   | ||     \     ||     \     ||  \     ||___   |
|__/\__/|__|_||__|\__\  __||__|\__\  __||__|\__\  __||______|
    |_____|      |_____|      |_____|      |_____|

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

Copyright (c) 2018 m0slevin, all rights reserved.
See license.txt for more information
===========================================================================*/
#pragma once

#include "mark3.h"

namespace Mark3
{
class LockGuard
{
public:
    LockGuard(Mutex* pclMutex);

    LockGuard(Mutex* pclMutex, uint32_t u32TimeoutMs_);

    ~LockGuard();

    bool isAcquired() { return m_bIsAcquired; }

private:
    bool    m_bIsAcquired;
    Mutex* m_pclMutex;
};
} // namespace Mark3
```

20.97 /home/moslevin/projects/m3-repo/kernel/src/public/mailbox.h File Reference

Mailbox + Envelope IPC Mechanism.

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

Classes

- class Mark3::Mailbox

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

Namespaces

- Mark3

20.97.1 Detailed Description

Mailbox + Envelope IPC Mechanism.

Definition in file mailbox.h.

20.98 mailbox.h

```
/*=============================================================================
     _____     _____     _____     _____
 ___|    _|__ __|_    |__ __|__   |__ __| __  |__  _____
|    \  /  | ||    \     ||     |     ||  |/ /     ||___   |
|     \/   | ||     \    ||     \     ||  |  \     ||___   |
|__/\__/|__|_||__|\__\ __||__|\__\ __||__|\__\ __||_______|
    |_____|      |_____|      |_____|      |_____|

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

Copyright (c) 2012 - 2019 m0slevin, all rights reserved.
See license.txt for more information
===========================================================================*/
#pragma once

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

namespace Mark3
{
class Mailbox
{
public:
    void* operator new(size_t sz, void* pv) { return reinterpret_cast<Mailbox*>(pv); }
    ~Mailbox();

    void Init(void* pvBuffer_, uint16_t u16BufferSize_, uint16_t u16ElementSize_);

    static Mailbox* Init(uint16_t u16BufferSize_, uint16_t u16ElementSize_);

    bool Send(void* pvData_);

    bool SendTail(void* pvData_);

    bool Send(void* pvData_, uint32_t u32TimeoutMS_);

    bool SendTail(void* pvData_, uint32_t u32TimeoutMS_);

    void Receive(void* pvData_);

    void ReceiveTail(void* pvData_);

    bool Receive(void* pvData_, uint32_t u32TimeoutMS_);

    bool ReceiveTail(void* pvData_, uint32_t u32TimeoutMS_);

    uint16_t GetFreeSlots(void)
    {
        const auto cs = CriticalGuard{};
        return m_u16Free;
```

```
    }

    bool IsFull(void) { return (GetFreeSlots() == 0); }
    bool IsEmpty(void) { return (GetFreeSlots() == m_u16Count); }

private:
    void* GetHeadPointer(void)
    {
        auto uAddr = reinterpret_cast<K_ADDR>(m_pvBuffer);
        uAddr += static_cast<K_ADDR>(m_u16ElementSize) * static_cast<K_ADDR>(m_u16Head);
        return reinterpret_cast<void*>(uAddr);
    }

    void* GetTailPointer(void)
    {
        auto uAddr = reinterpret_cast<K_ADDR>(m_pvBuffer);
        uAddr += static_cast<K_ADDR>(m_u16ElementSize) * static_cast<K_ADDR>(m_u16Tail);
        return reinterpret_cast<void*>(uAddr);
    }

    void CopyData(const void* src_, void* dst_, uint16_t len_)
    {
        auto* u8Src = reinterpret_cast<const uint8_t*>(src_);
        auto* u8Dst = reinterpret_cast<uint8_t*>(dst_);
        while (len_--) { *u8Dst++ = *u8Src++; }
    }

    void MoveTailForward(void)
    {
        m_u16Tail++;
        if (m_u16Tail == m_u16Count) {
            m_u16Tail = 0;
        }
    }

    void MoveHeadForward(void)
    {
        m_u16Head++;
        if (m_u16Head == m_u16Count) {
            m_u16Head = 0;
        }
    }

    void MoveTailBackward(void)
    {
        if (m_u16Tail == 0) {
            m_u16Tail = m_u16Count;
        }
        m_u16Tail--;
    }

    void MoveHeadBackward(void)
    {
        if (m_u16Head == 0) {
            m_u16Head = m_u16Count;
        }
        m_u16Head--;
    }

    bool Send_i(const void* pvData_, bool bTail_, uint32_t u32TimeoutMS_);

    bool Receive_i(void* pvData_, bool bTail_, uint32_t u32WaitTimeMS_);

    uint16_t m_u16Head;
    uint16_t m_u16Tail;

    uint16_t          m_u16Count;
    volatile uint16_t m_u16Free;

    uint16_t    m_u16ElementSize;
    const void* m_pvBuffer;

    Semaphore m_clRecvSem;
    Semaphore m_clSendSem;
};
} // namespace Mark3
```

20.99 /home/moslevin/projects/m3-repo/kernel/src/public/manual.h File Reference

20.99.1 Detailed Description

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

Definition in file manual.h.

20.100 manual.h

```
/*===========================================================================
     _____        _____        _____        _____
 ___|    _|__  __|_    |__  __|__   |__  __| __  |__  ______
|    \  /  | ||    \      ||     |     ||  |/ /     ||___   |
|     \/   | ||     \     ||     \     ||     \     ||___   |
|__/\__/|__|_||__|\__\  __||__|\__\  __||__|\__\  __||______|
    |_____|      |_____|      |_____|      |_____|

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

Copyright (c) 2012 - 2019 m0slevin, all rights reserved.
See license.txt for more information
===========================================================================*/


```

20.101 /home/moslevin/projects/m3-repo/kernel/src/public/mark3.h File Reference

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

```
#include "mark3cfg.h"
#include "threadport.h"
#include "criticalsection.h"
#include "criticalguard.h"
#include "kerneltypes.h"
#include "kerneldebug.h"
#include "ithreadport.h"
#include "kernelswi.h"
#include "kerneltimer.h"
#include "kernel.h"
#include "thread.h"
#include "timerlist.h"
#include "ksemaphore.h"
#include "mutex.h"
#include "lockguard.h"
#include "eventflag.h"
#include "message.h"
#include "notify.h"
#include "mailbox.h"
#include "readerwriter.h"
#include "condvar.h"
#include "atomic.h"
#include "profile.h"
#include "autoalloc.h"
#include "priomap.h"
#include "threadlist.h"
#include "threadlistlist.h"
#include "schedulerguard.h"
#include "coroutine.h"
#include "colist.h"
#include "cosched.h"
```

20.101.1 Detailed Description

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

Definition in file mark3.h.

20.102 mark3.h

```
/*===========================================================================
     _____        _____        _____        _____
 ___|    _|__  __|_    |__  __|__   |__  __| __  |__  ______
|    \  /  | ||    \      ||     |     ||  |/ /     ||___   |
|     \/   | ||     \     ||     \     ||     \     ||___   |
|__/\__/|__|_||__|\__\  __||__|\__\  __||__|\__\  __||______|
    |_____|      |_____|      |_____|      |_____|

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

Copyright (c) 2012 - 2019 m0slevin, all rights reserved.
See license.txt for more information
===========================================================================*/
#pragma once

#include "mark3cfg.h"
#include "threadport.h"
#include "criticalsection.h"
#include "criticalguard.h"

#include "kerneltypes.h"
#include "kerneldebug.h"

#include "ithreadport.h"
#include "kernelswi.h"
#include "kerneltimer.h"

#include "kernel.h"
#include "thread.h"
#include "timerlist.h"

#include "ksemaphore.h"
#include "mutex.h"
#include "lockguard.h"
#include "eventflag.h"
#include "message.h"
#include "notify.h"
#include "mailbox.h"
#include "readerwriter.h"
#include "condvar.h"

#include "atomic.h"

#include "profile.h"
#include "autoalloc.h"
#include "priomap.h"

#include "threadlist.h"
#include "threadlistlist.h"

#include "schedulerguard.h"

#include "coroutine.h"
#include "colist.h"
#include "cosched.h"

```

20.103 /home/moslevin/projects/m3-repo/kernel/src/public/mark3cfg.h File Reference

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.

```
#include "portcfg.h"
```

Macros

- #define KERNEL_DEBUG (0)
- #define KERNEL_STACK_CHECK (1)
- #define KERNEL_NAMED_THREADS (1)
- #define KERNEL_EVENT_FLAGS (1)
- #define KERNEL_CONTEXT_SWITCH_CALLOUT (1)
- #define KERNEL_THREAD_CREATE_CALLOUT (1)
- #define KERNEL_THREAD_EXIT_CALLOUT (1)
- #define KERNEL_ROUND_ROBIN (1)
- #define KERNEL_EXTENDED_CONTEXT (1)

    include CPU/Port specific configuration options

20.103.1 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.

Note: in the R7 and beyond version of the kernel, all options are enabled by default. As a result, the only configuration options presented are now located within the architecture-specific "portcfg.h".

Definition in file mark3cfg.h.

20.103.2 Macro Definition Documentation

20.103.2.1 KERNEL_CONTEXT_SWITCH_CALLOUT

```
#define KERNEL_CONTEXT_SWITCH_CALLOUT (1)
```

When enabled, this feature allows a user to define a callback to be executed whenever a context switch occurs. Enabling this provides a means for a user to track thread statistics, but it does result in additional overhead during a context switch.

Definition at line 70 of file mark3cfg.h.

20.103.2.2 KERNEL_DEBUG

```
#define KERNEL_DEBUG (0)
```

Enable kernel asserts at runtime.

Definition at line 30 of file mark3cfg.h.

20.103.2.3 KERNEL_EVENT_FLAGS

```
#define KERNEL_EVENT_FLAGS (1)
```

This flag enables the event-flags synchronization object. This feature allows threads to be blocked, waiting on specific condition bits to be set or cleared on an EventFlag object.

While other synchronization objects are enabled by default, this one is configurable because it impacts the Thread object's member data.

Definition at line 62 of file mark3cfg.h.

20.103.2.4 KERNEL_EXTENDED_CONTEXT

```
#define KERNEL_EXTENDED_CONTEXT (1)
```

include CPU/Port specific configuration options

Provide a special data pointer in the thread object, which may be used to add additional context to a thread. Typically this would be used to implement thread-local-storage.

Definition at line 97 of file mark3cfg.h.

20.103.2.5 KERNEL_NAMED_THREADS

```
#define KERNEL_NAMED_THREADS (1)
```

Enabling this provides the Thread::SetName() and Thread::GetName() methods, allowing for each thread to be named with a null-terminated const char∗ string.

Note: the string passed to Thread::SetName() must persist for the lifetime of the thread

Definition at line 52 of file mark3cfg.h.

20.103.2.6 KERNEL_ROUND_ROBIN

```
#define KERNEL_ROUND_ROBIN (1)
```

Enable round-robin scheduling within each priority level. When selected, this results in a small performance hit during context switching and in the system tick handler, as a special software timer is used to manage the running thread's quantum. Can be disabled to optimize performance if not required.

Definition at line 90 of file mark3cfg.h.

20.103.2.7 KERNEL_STACK_CHECK

```
#define KERNEL_STACK_CHECK (1)
```

Perform stack-depth checks on threads at each context switch, which is useful in detecting stack overflows / near overflows. Near-overflow detection uses thresholds defined in the target's portcfg.h. Enabling this also adds the Thread::GetStackSlack() method, which allows a thread's stack to be profiled on-demand.

Note: When enabled, the additional stack checks result in a performance hit to context switches and thread initialization.

Definition at line 43 of file mark3cfg.h.

20.103.2.8 KERNEL_THREAD_CREATE_CALLOUT

```
#define KERNEL_THREAD_CREATE_CALLOUT (1)
```

This feature provides a user-defined kernel callback that is executed whenever a thread is started.

Definition at line 76 of file mark3cfg.h.

20.103.2.9 KERNEL_THREAD_EXIT_CALLOUT

```
#define KERNEL_THREAD_EXIT_CALLOUT (1)
```

This feature provides a user-defined kernel callback that is executed whenever a thread is terminated.

Definition at line 82 of file mark3cfg.h.

20.104 mark3cfg.h

```
00001 /*===========================================================================
00002      _____        _____        _____        _____
00003  ___|    _|__  __|_    |__  __|__   |__  __| __  |__  ______
00004 |    \  /  | ||    \      ||     |     ||  |/ /     ||___   |
00005 |     \/   | ||     \     ||     \     ||     \     ||___   |
00006 |__/\__/|__|_||__|\__\  __||__|\__\  __||__|\__\  __||______|
00007     |_____|      |_____|      |_____|      |_____|
00008
00009 --[Mark3 Realtime Platform]--------------------------------------------------
00010
00011 Copyright (c) 2012 - 2019 m0slevin, all rights reserved.
00012 See license.txt for more information
00013 ===========================================================================*/
00025 #pragma once
00026
00030 #define KERNEL_DEBUG (0)
00031
00043 #define KERNEL_STACK_CHECK (1)
00044
00052 #define KERNEL_NAMED_THREADS (1)
00053
00062 #define KERNEL_EVENT_FLAGS (1)
00063
00070 #define KERNEL_CONTEXT_SWITCH_CALLOUT (1)
00071
00076 #define KERNEL_THREAD_CREATE_CALLOUT (1)
00077
00082 #define KERNEL_THREAD_EXIT_CALLOUT (1)
00083
00090 #define KERNEL_ROUND_ROBIN (1)
00091
00097 #define KERNEL_EXTENDED_CONTEXT (1)
00098
00099 #include "portcfg.h"
```

20.105 /home/moslevin/projects/m3-repo/kernel/src/public/message.h File Reference

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.

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

Classes

- class Mark3::Message

  the Message class. This object provides threadsafe message-based IPC services based on exchange of objects containing a data pointer and minimal application-defined metadata. Messages are to be allocated/produced by the sender, and deallocated/consumed by the receiver.
- class Mark3::MessagePool

  The MessagePool Class The MessagePool class implements a simple allocator for message objects exchanged between threads. The sender allocates (pop's) messages, then sends them to the receiver. Upon receipt, it is the receiver's responsibility to deallocate (push) the message back to the pool.
- class Mark3::MessageQueue

  The MessageQueue class. Implements a mechanism used to send/receive data between threads. Allows threads to block, waiting for messages to be sent from other contexts.

Namespaces

- Mark3

20.105.1 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.

20.105.2 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( nullptr );

        // 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.

20.106 message.h

```
/*===========================================================================
     _____       _____       _____       _____
 ___|    _|__ __|_    |__ __|__   |__ __| __  |__ ______
|    \  /  | ||    \    ||     |    ||  |/ /    ||___   |
|     \/   | ||     \   ||     \    ||     \    ||___   |
|__/\__/|__|_||__|\__\ __||__|\__\ __||__|\__\ __||______|
    |_____|      |_____|      |_____|      |_____|

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

Copyright (c) 2012 - 2019 m0slevin, all rights reserved.
See license.txt for more information
=========================================================================== */
#pragma once

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

#include "ll.h"
#include "ksemaphore.h"
#include "timerlist.h"

namespace Mark3
{
//---------------------------------------------------------------------------
class Message : public TypedLinkListNode<Message>
{
public:
    void* operator new(size_t sz, void* pv) { return reinterpret_cast<Message*>(pv); }
```

```
    void Init()
    {
        ClearNode();
        m_pvData  = nullptr;
        m_u16Code = 0;
    }

    void SetData(void* pvData_) { m_pvData = pvData_; }
    void* GetData() { return m_pvData; }
    void SetCode(uint16_t u16Code_) { m_u16Code = u16Code_; }
    uint16_t GetCode() { return m_u16Code; }

private:
    void* m_pvData;

    uint16_t m_u16Code;
};

//---------------------------------------------------------------------------
class MessagePool
{
public:
    void* operator new(size_t sz, void* pv) { return (MessagePool*)pv; }
    ~MessagePool() {}
    void Init();

    void Push(Message* pclMessage_);

    Message* Pop();

    Message* GetHead();

private:
    TypedDoubleLinkList<Message> m_clList;
};

//---------------------------------------------------------------------------
class MessageQueue
{
public:
    void* operator new(size_t sz, void* pv) { return (MessageQueue*)pv; }
    ~MessageQueue() {}

    void Init();

    Message* Receive();

    Message* Receive(uint32_t u32TimeWaitMS_);

    void Send(Message* pclSrc_);

    uint16_t GetCount();

private:
    Message* Receive_i(uint32_t u32TimeWaitMS_);

    Semaphore m_clSemaphore;

    TypedDoubleLinkList<Message> m_clLinkList;
};
} // namespace Mark3
```

20.107 /home/moslevin/projects/m3-repo/kernel/src/public/mutex.h File Reference

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.

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

Classes

- class Mark3::Mutex

  The Mutex Class. Class providing Mutual-exclusion locks, based on BlockingObject.

Namespaces

- Mark3

20.107.1 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.

20.107.2 Initializing

Initializing a mutex object by calling:

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

20.107.3 Resource protection example

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

Definition in file mutex.h.

20.108 mutex.h

```
/*===========================================================================
     _____        _____        _____        _____
 ___|    _|__  __|_    |__  __|__   |__  __| __  |__  ______
|    \  /  | ||    \      ||     |     ||  |/ /     ||___   |
|     \/   | ||     \     ||     \     ||     \     ||___   |
|__/\__/|__|_||__|\__\  __||__|\__\  __||__|\__\  __||______|
    |_____|      |_____|      |_____|      |_____|

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

Copyright (c) 2012 - 2019 m0slevin, all rights reserved.
See license.txt for more information
=========================================================================== */
#pragma once

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

```

```
#include "blocking.h"
namespace Mark3
{
//------------------------------------------------------------------------
class Mutex : public BlockingObject
{
public:
    void* operator new(size_t sz, void* pv) { return reinterpret_cast<Mutex*>(pv); };
    ~Mutex();
    void Init(bool bRecursive_ = true);
    void Claim();
    bool Claim(uint32_t u32WaitTimeMS_);
    void WakeMe(Thread* pclOwner_);
    void Release();
private:
    uint8_t WakeNext();
    bool Claim_i(uint32_t u32WaitTimeMS_);
    uint8_t        m_u8Recurse;
    bool           m_bReady;
    bool           m_bRecursive;
    PORT_PRIO_TYPE m_uMaxPri;
    Thread*        m_pclOwner;
};
} // namespace Mark3
```

20.109 /home/moslevin/projects/m3-repo/kernel/src/public/notify.h File Reference

Lightweight thread notification - blocking object.

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

Classes

- class Mark3::Notify

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

Namespaces

- Mark3

20.109.1 Detailed Description

Lightweight thread notification - blocking object.

Definition in file notify.h.

20.110 notify.h

```
/*===========================================================================
     _____        _____        _____        _____
 ___|    _|__  __|_    |__  __|__   |__  __| __  |__  ______
|    \  /  | ||    \      ||     |     ||  |/ /     ||___   |
|     \/   | ||     \     ||     \     ||     \     ||___   |
|__/\__/|__|_||__|\__\  __||__|\__\  __||__|\__\  __||______|
    |_____|      |_____|      |_____|      |_____|

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

Copyright (c) 2012 - 2019 m0slevin, all rights reserved.
See license.txt for more information
===========================================================================*/
#pragma once

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

namespace Mark3
{
class Notify : public BlockingObject
{
public:
    void* operator new(size_t sz, void* pv) { return reinterpret_cast<Notify*>(pv); };
    ~Notify();

    void Init(void);

    void Signal(void);

    void Wait(bool* pbFlag_);

    bool Wait(uint32_t u32WaitTimeMS_, bool* pbFlag_);

    void WakeMe(Thread* pclChosenOne_);

private:
    bool m_bPending;
};
} // namespace Mark3
```

20.111 /home/moslevin/projects/m3-repo/kernel/src/public/paniccodes.h File Reference

Defines the reason codes thrown when a kernel panic occurs.

Macros

- #define PANIC_ASSERT_FAILED (1)
- #define PANIC_LIST_UNLINK_FAILED (2)
- #define PANIC_STACK_SLACK_VIOLATED (3)
- #define PANIC_AUTO_HEAP_EXHAUSTED (4)
- #define PANIC_POWERMAN_EXHAUSTED (5)
- #define PANIC_NO_READY_THREADS (6)
- #define PANIC_RUNNING_THREAD_DESCOPED (7)
- #define PANIC_ACTIVE_SEMAPHORE_DESCOPED (8)
- #define PANIC_ACTIVE_MUTEX_DESCOPED (9)
- #define PANIC_ACTIVE_EVENTFLAG_DESCOPED (10)
- #define PANIC_ACTIVE_NOTIFY_DESCOPED (11)
- #define PANIC_ACTIVE_MAILBOX_DESCOPED (12)
- #define PANIC_ACTIVE_TIMER_DESCOPED (13)
- #define PANIC_ACTIVE_COROUTINE_DESCOPED (14)

20.111.1 Detailed Description

Defines the reason codes thrown when a kernel panic occurs.

Definition in file paniccodes.h.

20.111.2 Macro Definition Documentation

20.111.2.1 PANIC_ACTIVE_COROUTINE_DESCOPED

```
#define PANIC_ACTIVE_COROUTINE_DESCOPED (14)
```

Definition at line 35 of file paniccodes.h.

20.111.2.2 PANIC_ACTIVE_EVENTFLAG_DESCOPED

```
#define PANIC_ACTIVE_EVENTFLAG_DESCOPED (10)
```

Definition at line 31 of file paniccodes.h.

20.111.2.3 PANIC_ACTIVE_MAILBOX_DESCOPED

```
#define PANIC_ACTIVE_MAILBOX_DESCOPED (12)
```

Definition at line 33 of file paniccodes.h.

20.111.2.4 PANIC_ACTIVE_MUTEX_DESCOPED

```
#define PANIC_ACTIVE_MUTEX_DESCOPED (9)
```

Definition at line 30 of file paniccodes.h.

20.111.2.5 PANIC_ACTIVE_NOTIFY_DESCOPED

```
#define PANIC_ACTIVE_NOTIFY_DESCOPED (11)
```

Definition at line 32 of file paniccodes.h.

20.111.2.6 PANIC_ACTIVE_SEMAPHORE_DESCOPED

```
#define PANIC_ACTIVE_SEMAPHORE_DESCOPED (8)
```

Definition at line 29 of file paniccodes.h.

20.111.2.7 PANIC_ACTIVE_TIMER_DESCOPED

```
#define PANIC_ACTIVE_TIMER_DESCOPED (13)
```

Definition at line 34 of file paniccodes.h.

20.111.2.8 PANIC_ASSERT_FAILED

```
#define PANIC_ASSERT_FAILED (1)
```

Definition at line 22 of file paniccodes.h.

20.111.2.9 PANIC_AUTO_HEAP_EXHAUSTED

```
#define PANIC_AUTO_HEAP_EXHAUSTED (4)
```

Definition at line 25 of file paniccodes.h.

20.111.2.10 PANIC_LIST_UNLINK_FAILED

```
#define PANIC_LIST_UNLINK_FAILED (2)
```

Definition at line 23 of file paniccodes.h.

20.111.2.11 PANIC_NO_READY_THREADS

```
#define PANIC_NO_READY_THREADS (6)
```

Definition at line 27 of file paniccodes.h.

20.111.2.12 PANIC_POWERMAN_EXHAUSTED

```
#define PANIC_POWERMAN_EXHAUSTED (5)
```

Definition at line 26 of file paniccodes.h.

20.111.2.13 PANIC_RUNNING_THREAD_DESCOPED

```
#define PANIC_RUNNING_THREAD_DESCOPED (7)
```

Definition at line 28 of file paniccodes.h.

20.111.2.14 PANIC_STACK_SLACK_VIOLATED

```
#define PANIC_STACK_SLACK_VIOLATED (3)
```

Definition at line 24 of file paniccodes.h.

20.112 paniccodes.h

```
00001 /*============================================================================
00002      _____     _____     _____     _____
00003  ___|    _|__ __|_    |__ __|__   |__ __| __  |__ ______
00004 |    \  /  | ||    \    ||     |     ||  |/ /    ||___   |
00005 |     \/   | ||     \   ||     \     ||     \    ||___   |
00006 |__/\__/|__|_||__|\__\ __||__|\__\ __||__|\__\ __||______|
00007     |_____|      |_____|      |_____|      |_____|
00008
00009 --[Mark3 Realtime Platform]--------------------------------------------------
00010
00011 Copyright (c) 2012 - 2019 m0slevin, all rights reserved.
00012 See license.txt for more information
00013 ===========================================================================*/
00020 #pragma once
00021
00022 #define PANIC_ASSERT_FAILED (1)
00023 #define PANIC_LIST_UNLINK_FAILED (2)
00024 #define PANIC_STACK_SLACK_VIOLATED (3)
00025 #define PANIC_AUTO_HEAP_EXHAUSTED (4)
00026 #define PANIC_POWERMAN_EXHAUSTED (5)
00027 #define PANIC_NO_READY_THREADS (6)
00028 #define PANIC_RUNNING_THREAD_DESCOPED (7)
00029 #define PANIC_ACTIVE_SEMAPHORE_DESCOPED (8)
00030 #define PANIC_ACTIVE_MUTEX_DESCOPED (9)
00031 #define PANIC_ACTIVE_EVENTFLAG_DESCOPED (10)
00032 #define PANIC_ACTIVE_NOTIFY_DESCOPED (11)
00033 #define PANIC_ACTIVE_MAILBOX_DESCOPED (12)
00034 #define PANIC_ACTIVE_TIMER_DESCOPED (13)
00035 #define PANIC_ACTIVE_COROUTINE_DESCOPED (14)
```

20.113 /home/moslevin/projects/m3-repo/kernel/src/public/priomap.h File Reference

Priority map data structure.

```
#include "kerneltypes.h"
#include "mark3cfg.h"
#include "priomapl1.h"
#include "priomapl2.h"
```

Namespaces

- Mark3

Typedefs

- using Mark3::PriorityMap = PriorityMapL1< PORT_PRIO_TYPE, KERNEL_NUM_PRIORITIES >

20.113.1 Detailed Description

Priority map data structure.

Definition in file priomap.h.

20.114 priomap.h

```
/*===========================================================================
     _____       _____       _____       _____
 ___|    _|__  __|_    |__  __|__   |__  __| __  |__  _____
|    \  /  | ||    \     ||     |     ||  |/ /     ||___   |
|     \/   | ||     \    ||     \     ||     \     ||___   |
|__/\__/|__|_||__|\__\  __||__|\__\  __||__|\__\  __||_______|
    |_____|      |_____|      |_____|      |_____|

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

Copyright (c) 2012 - 2019 m0slevin, all rights reserved.
See license.txt for more information
===========================================================================*/
#pragma once

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

#include "priomapl1.h"
#include "priomapl2.h"

namespace Mark3 {
#if KERNEL_NUM_PRIORITIES <= (PORT_PRIO_MAP_WORD_SIZE * 8u)
using PriorityMap =
    PriorityMapL1<PORT_PRIO_TYPE, KERNEL_NUM_PRIORITIES>;
#else
using PriorityMap =
    PriorityMapL2<PORT_PRIO_TYPE, KERNEL_NUM_PRIORITIES>;
#endif
} // namespace Mark3
```

20.115 /home/moslevin/projects/m3-repo/kernel/src/public/priomapl1.h File Reference

1-Level bitmap allocator template-class used for scheduler implementation

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

Classes

- class Mark3::PriorityMapL1< T, C >

  The PriorityMapL1 class This class implements a priority bitmap data structure. Each bit in the objects internal storage represents a priority. When a bit is set, it indicates that something is scheduled at the bit's corresponding priority, when a bit is clear it indicates that no entities are scheduled at that priority. This object provides the fundamental logic required to implement efficient priority-based scheduling for the thread + coroutine schedulers in the kernel.

Namespaces

- Mark3

20.115.1 Detailed Description

1-Level bitmap allocator template-class used for scheduler implementation

Definition in file priomapl1.h.

20.116 priomapl1.h

```
/*===========================================================================
     _____        _____        _____        _____
 ___|    _|__  __|_    |__  __|__   |__  __| __  |__  ______
|    \  /  | ||    \      ||     |     ||  |/ /     ||___   |
|     \/   | ||     \     ||     \     ||  \        ||___   |
|__/\__/|__|_||__|\__\  __||__|\__\  __||__|\__\  __||______|
    |_____|      |_____|      |_____|      |_____|

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

Copyright (c) 2012 - 2019 m0slevin, all rights reserved.
See license.txt for more information
===========================================================================*/
#pragma once

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

namespace Mark3
{
//---------------------------------------------------------------------------
template <typename T, size_t C>
class PriorityMapL1
{
public:
    PriorityMapL1()
    {
        m_uXPriorityMap = 0;
    }

    void Set(T uXPrio_)
    {
        auto uXPrioBit = PrioBit(uXPrio_);
        m_uXPriorityMap |= (1 << uXPrioBit);
    }

    void Clear(T uXPrio_)
    {
        auto uXPrioBit = PrioBit(uXPrio_);
        m_uXPriorityMap &= ~(1 << uXPrioBit);
    }

    T HighestPriority(void)
    {
        auto uXPrio = PriorityFromBitmap(m_uXPriorityMap);
        return uXPrio;
    }

```

```
private:
    static inline T PrioBit(T prio) { return prio & m_uXPrioMapBitMask; }

    static inline T PrioMapWordIndex(T prio) { return prio >> m_uXPrioMapWordShift; }

    static inline T PriorityFromBitmap(T uXPrio_)
    {
#if PORT_USE_HW_CLZ
        // Support hardware-accelerated Count-leading-zeros instruction
        return m_uXPrioMapBits - PORT_CLZ(uXPrio_);
#else
        // Default un-optimized count-leading zeros operation
        T uXMask  = 1 << (m_uXPrioMapBits - 1);
        auto          u8Zeros = T { 0 };

        while (uXMask) {
            if (uXMask & uXPrio_) {
                return (m_uXPrioMapBits - u8Zeros);
            }

            uXMask >>= 1;
            u8Zeros++;
        }
        return 0;
#endif
    }

    static constexpr size_t m_uXPrioMapShiftLUT[9] = {0, 3, 4, 0, 5, 0, 0, 0, 6};
    static constexpr auto m_uXPrioMapWordShift = T { m_uXPrioMapShiftLUT[sizeof(T)] };
    static constexpr auto m_uXPrioMapBits     = T { 8 * sizeof(T) };
    static constexpr auto m_uXPrioMapBitMask = T { (1 << m_uXPrioMapWordShift) - 1 };

    T m_uXPriorityMap;
};
} // namespace Mark3
```

20.117 /home/moslevin/projects/m3-repo/kernel/src/public/priomapl2.h File Reference

2-Level priority allocator template-class used for scheduler implementation

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

Classes

- class Mark3::PriorityMapL2< T, C >

  The PriorityMapL2 class This class implements a priority bitmap data structure. Each bit in the objects internal storage represents a priority. When a bit is set, it indicates that something is scheduled at the bit's corresponding priority, when a bit is clear it indicates that no entities are scheduled at that priority. This object provides the fundamental logic required to implement efficient priority-based scheduling for the thread + coroutine schedulers in the kernel.

Namespaces

- Mark3

20.117.1 Detailed Description

2-Level priority allocator template-class used for scheduler implementation

Definition in file priomapl2.h.

20.118 priomapl2.h

```
/*===========================================================================
     _____        _____        _____        _____
 ___|    _|__  __|_    |__  __|__   |__  __| __  |__  ______
|    \  /  | ||    \      ||     |     ||  |/ /     ||___   |
|     \/   | ||     \     ||     \     ||     \     ||___   |
|__/\__/|__|_||__|\__\  __||__|\__\  __||__|\__\  __||______|
    |_____|      |_____|      |_____|      |_____|

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

Copyright (c) 2012 - 2019 m0slevin, all rights reserved.
See license.txt for more information
===========================================================================*/
#pragma once

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

namespace Mark3
{
//---------------------------------------------------------------------------
template <typename T, size_t C>
class PriorityMapL2
{
public:
    PriorityMapL2()
    {
        m_uXPriorityMapL2 = 0;
        for (auto i = PORT_PRIO_TYPE{0}; i < m_uXPrioMapNumWords; i++) {
m_auXPriorityMap[i] = 0; }
    }

    void Set(T uXPrio_)
    {
        auto uXPrioBit = PrioBit(uXPrio_);
        auto uXWordIdx = PrioMapWordIndex(uXPrio_);

        m_auXPriorityMap[uXWordIdx] |= (1 << uXPrioBit);
        m_uXPriorityMapL2 |= (1 << uXWordIdx);
    }

    void Clear(T uXPrio_)
    {
        auto uXPrioBit = PrioBit(uXPrio_);
        auto uXWordIdx = PrioMapWordIndex(uXPrio_);

        m_auXPriorityMap[uXWordIdx] &= ~(1 << uXPrioBit);
        if (!m_auXPriorityMap[uXWordIdx]) {
            m_uXPriorityMapL2 &= ~(1 << uXWordIdx);
        }
    }

    T HighestPriority(void)
    {
        auto uXMapIdx = PriorityFromBitmap(m_uXPriorityMapL2);
        if (!uXMapIdx) {
            return 0;
        }
        uXMapIdx--;
        auto uXPrio = PriorityFromBitmap(m_auXPriorityMap[uXMapIdx]);
        uXPrio += (uXMapIdx * m_uXPrioMapBits);
        return uXPrio;
    }

private:
    static inline T PrioBit(T prio) { return prio & m_uXPrioMapBitMask; }

    static inline T PrioMapWordIndex(T prio) { return prio >>
m_uXPrioMapWordShift; }

    static inline T PriorityFromBitmap(T uXPrio_)
    {
#if PORT_USE_HW_CLZ
        // Support hardware-accelerated Count-leading-zeros instruction
        return m_uXPrioMapBits - PORT_CLZ(uXPrio_);
#else
        // Default un-optimized count-leading zeros operation
        T uXMask  = 1 << (m_uXPrioMapBits - 1);
        auto           u8Zeros = T { 0 };

        while (uXMask) {
            if (uXMask & uXPrio_) {
                return (m_uXPrioMapBits - u8Zeros);
```

```
            }

            uXMask >>= 1;
            u8Zeros++;
        }
        return 0;
#endif
    }

    static constexpr size_t m_uXPrioMapShiftLUT[9] = {0, 3, 4, 0, 5, 0, 0, 0, 6};
    static constexpr auto m_uXPrioMapWordShift = T { m_uXPrioMapShiftLUT[sizeof(T)] };
    static constexpr auto m_uXPrioMapBits    = T { 8 * sizeof(T) };
    static constexpr auto m_uXPrioMapBitMask = T { (1 <<
m_uXPrioMapWordShift) - 1 };

    // Required size of the bitmap array in words
    static constexpr auto m_uXPrioMapNumWords
        = T { (C + (m_uXPrioMapBits - 1)) / m_uXPrioMapBits };

    T m_auXPriorityMap[m_uXPrioMapNumWords];
    T m_uXPriorityMapL2;
};
} // namespace Mark3
```

20.119 /home/moslevin/projects/m3-repo/kernel/src/public/profile.h File Reference

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.

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

Classes

- class Mark3::ProfileTimer

  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.

Namespaces

- Mark3

20.119.1 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.

20.120 profile.h

```
/*===========================================================================
     _____         _____         _____         _____
 ___|    _|__  __|_    |__  __|__    |__  __| __  |__  ______
|    \  /  | ||    \      ||     |      ||  |/ /     ||___   |
|     \/   | ||     \     ||     \      ||     \     ||___   |
|__/\__/|__|_||__|\__\  __||__|\__\  __||__|\__\  __||______|
    |_____|      |_____|      |_____|      |_____|

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

Copyright (c) 2012 - 2019 m0slevin, all rights reserved.
See license.txt for more information
=========================================================================== */
#pragma once

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

namespace Mark3
{
class ProfileTimer
{
public:
    void Init();

    void Start();

    void Stop();

    uint32_t GetAverage();

    uint32_t GetCurrent();

private:
    uint32_t m_u32StartTicks;
    uint32_t m_u32CurrentIteration;
    uint32_t m_u32Cumulative;
    uint16_t m_u16Iterations;
    bool     m_bActive;
};
} // namespace Mark3
```

20.121 /home/moslevin/projects/m3-repo/kernel/src/public/profiling_results.h File Reference

20.122 profiling_results.h

20.123 /home/moslevin/projects/m3-repo/kernel/src/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 Mark3::Quantum

  The Quantum Class. Static-class used to implement Thread quantum functionality, which is fundamental to round-robin thread scheduling.

Namespaces

- Mark3

20.123.1 Detailed Description

Thread Quantum declarations for Round-Robin Scheduling.

Definition in file quantum.h.

20.124 quantum.h

```
/*===========================================================================
     _____        _____        _____        _____
 ___|    _|__  __|_    |__  __|__   |__  __| __  |__  ______
|    \  /  | ||    \      ||     |     ||  |/ /     ||___   |
|     \/   | ||     \     ||     \     ||     \     ||___   |
|__/\__/|__|_||__|\__\  __||__|\__\  __||__|\__\  __||______|
    |_____|      |_____|      |_____|      |_____|

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

Copyright (c) 2012 - 2019 m0slevin, all rights reserved.
See license.txt for more information
=========================================================================== */
#pragma once

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

#include "thread.h"
#include "timer.h"
#include "timerlist.h"
#include "timerscheduler.h"

#if KERNEL_ROUND_ROBIN
namespace Mark3
{
class Timer;

class Quantum
{
```

```
public:
    static void Init();

    static void SetInTimer();

    static void ClearInTimer();

    static void Update(Thread* pclTargetThread_);

    static void SetTimerThread(Thread* pclTimerThread_) {
m_pclTimerThread = pclTimerThread_; }

    static void Cancel();

private:
    static Thread*  m_pclActiveThread;
    static Thread*  m_pclTimerThread;
    static uint16_t m_u16TicksRemain;
    static bool     m_bInTimer;
};
} // namespace Mark3
#endif // #if KERNEL_ROUND_ROBIN
```

20.125 /home/moslevin/projects/m3-repo/kernel/src/public/readerwriter.h File Reference

Reader-Writer lock implementation.

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

Classes

- class Mark3::ReaderWriterLock

    The ReaderWriterLock class. This class implements an object that marshalls access to a resource based on the intended usage of the resource. A reader-writer lock permits multiple concurrent read access, or single-writer access to a resource. If the object holds a write lock, other writers, and all readers will block until the writer is finished. If the object holds reader locks, all writers will block until all readers are finished before the first writer can take ownership of the resource. This is based upon lower-level synchronization primatives, and is somewhat more heavyweight than primative synchronization types.

Namespaces

- Mark3

20.125.1 Detailed Description

Reader-Writer lock implementation.

Definition in file readerwriter.h.

20.126 readerwriter.h

```
/*===========================================================================
     _____        _____        _____        _____
 ___|    _|__  __|_    |__  __|__   |__  __| __  |__  ______
|    \  /  | ||    \      ||     |     ||  |/ /     ||___   |
|     \/   | ||     \     ||     \     ||     \     ||___   |
|__/\__/|__|_||__|\__\  __||__|\__\  __||__|\__\  __||______|
    |_____|      |_____|      |_____|      |_____|

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

Copyright (c) 2012 - 2019 m0slevin, all rights reserved.
See license.txt for more information
===========================================================================*/
#pragma once

#include "mark3cfg.h"
#include "blocking.h"
#include "mutex.h"

namespace Mark3
{
class ReaderWriterLock
{
public:
    void* operator new(size_t sz, void* pv) { return reinterpret_cast<
ReaderWriterLock*>(pv); }

    void Init();

    void AcquireReader();

    bool AcquireReader(uint32_t u32TimeoutMs_);

    void ReleaseReader();

    void AcquireWriter();

    bool AcquireWriter(uint32_t u32TimeoutMs_);

    void ReleaseWriter();

private:
    bool AcquireReader_i(uint32_t u32TimeoutMs_);
    bool AcquireWriter_i(uint32_t u32TimeoutMs_);

    Mutex   m_clGlobalMutex;
    Mutex   m_clReaderMutex;
    uint8_t m_u8ReadCount;
};

} // namespace Mark3
```

20.127 /home/moslevin/projects/m3-repo/kernel/src/public/scheduler.h File Reference

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.

```
#include "kerneltypes.h"
#include "thread.h"
#include "ithreadport.h"
#include "priomap.h"
```

Classes

- class Mark3::Scheduler

  The Scheduler Class. This class provides priority-based round-robin Thread scheduling for all active threads managed by the kernel.

Namespaces

- Mark3

Variables

- Mark3::Thread ∗ g_pclNext
- Mark3::Thread ∗ g_pclCurrent

20.127.1 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.

20.127.2 Variable Documentation

20.127.2.1 g_pclCurrent

```
Mark3::Thread* g_pclCurrent
```

Definition at line 25 of file scheduler.cpp.

20.127.2.2 g_pclNext

```
Mark3::Thread* g_pclNext
```

Definition at line 24 of file scheduler.cpp.

20.128 scheduler.h

```
/*===========================================================================
     _____       _____       _____       _____
 ___|    _|__ __|_    |__ __|__   |__ __| __  |__  ______
|    \  /  | ||    \    ||     |     ||  |/ /    ||___   |
|     \/   | ||     \   ||     \     ||     \    ||___   |
|__/\__/|__|_||__|\__\ __||__|\__\ __||__|\__\ __||______|
    |_____|      |_____|      |_____|      |_____|

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

Copyright (c) 2012 - 2019 m0slevin, all rights reserved.
See license.txt for more information
=========================================================================== */
#pragma once

#include "kerneltypes.h"
#include "thread.h"
#include "ithreadport.h"
#include "priomap.h"

extern Mark3::Thread* g_pclNext;
extern Mark3::Thread* g_pclCurrent;

namespace Mark3
{
//---------------------------------------------------------------------------
class Scheduler
{
public:
    static void Init();

    static void Schedule();

    static void Add(Thread* pclThread_);

    static void Remove(Thread* pclThread_);

    static bool SetScheduler(bool bEnable_);

    static Thread* GetCurrentThread() { return
g_pclCurrent; }
    static volatile Thread* GetNextThread() { return
g_pclNext; }
    static ThreadList* GetThreadList(PORT_PRIO_TYPE uXPriority_) {
return &m_aclPriorities[uXPriority_]; }
    static ThreadList* GetStopList() { return &m_clStopList; }
    static bool IsEnabled() { return m_bEnabled; }
    static void QueueScheduler() { m_bQueuedSchedule = true; }

private:
    static constexpr auto m_uNumPriorities = size_t {
KERNEL_NUM_PRIORITIES };

    static bool m_bEnabled;

    static bool m_bQueuedSchedule;

    static ThreadList m_clStopList;

    static ThreadList m_aclPriorities[m_uNumPriorities];

    static PriorityMap m_clPrioMap;
};
} // namespace Mark3
```

20.129 /home/moslevin/projects/m3-repo/kernel/src/public/schedulerguard.h File Reference

RAII Scheduler Locking.

```
#include "mark3cfg.h"
#include "scheduler.h"
```

Classes

- class Mark3::SchedulerGuard

  The SchedulerGuard class This class implements RAII-based control of the scheduler's global state. Upon object construction, the scheduler's state is cached locally and the scheduler is disabled (if not already disabled). Upon object destruction, the scheduler's previous state is restored. This object is interrupt-safe, although it has no effect when called from an interrupt given that interrupts are inherently higher-priority than threads.

Namespaces

- Mark3

20.129.1 Detailed Description

RAII Scheduler Locking.

Definition in file schedulerguard.h.

20.130 schedulerguard.h

```
/*===========================================================================
     _____       _____       _____       _____
 ___|    _|__  __|_    |__  __|__   |__  __| __  |__  ______
|    \  /  | ||    \      ||     |     ||  |/ /    ||___   |
|     \/   | ||     \     ||     \     ||    \     ||___   |
|__/\__/|__|_||__|\__\  __||__|\__\  __||__|\__\  __||_______|
    |_____|      |_____|      |_____|      |_____|

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

Copyright (c) 2012 - 2019 m0slevin, all rights reserved.
See license.txt for more information
===========================================================================*/
#pragma once

#include "mark3cfg.h"
#include "scheduler.h"

namespace Mark3 {
class SchedulerGuard {
public:
    SchedulerGuard()
    {
        m_bSchedState = Scheduler::SetScheduler(false);
    }

    ~SchedulerGuard()
    {
        Scheduler::SetScheduler(m_bSchedState);
    }

private:
    bool m_bSchedState;
};

} // namespace Mark3
```

20.131 /home/moslevin/projects/m3-repo/kernel/src/public/sizeprofile.h File Reference

20.132 sizeprofile.h

```
#pragma once

```

20.133 /home/moslevin/projects/m3-repo/kernel/src/public/thread.h File Reference

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.

```
#include "kerneltypes.h"
#include "mark3cfg.h"
#include "ll.h"
#include "threadlist.h"
#include "scheduler.h"
#include "ithreadport.h"
#include "quantum.h"
#include "autoalloc.h"
#include "priomap.h"
```

Classes

- class Mark3::Thread

  The Thread Class. This object providing the fundamental thread control data structures and functions that define a single thread of execution in the Mark3 operating system. It is the fundamental data type used to provide multitasking support in the kernel.

Namespaces

- Mark3

Typedefs

- using Mark3::ThreadCreateCallout = void(∗)(Thread ∗pclThread_)
- using Mark3::ThreadExitCallout = void(∗)(Thread ∗pclThread_)
- using Mark3::ThreadContextCallout = void(∗)(Thread ∗pclThread_)

20.133.1 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.

20.134 thread.h

```
/*=============================================================================
     _____     _____     _____     _____
 ___|    _|__ __|_    |__ __|__   |__ __| __  |__ ______
|    \  /  | ||    \    ||     |     ||  |/ /    ||___   |
|     \/   | ||     \   ||     \     ||     \    ||___   |
|__/\__/|__|_||__|\__\ __||__|\__\ __||__|\__\ __||______|
    |_____|      |_____|     |_____|     |_____|

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

Copyright (c) 2012 - 2019 m0slevin, all rights reserved.
See license.txt for more information
=========================================================================== */
#pragma once

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

#include "ll.h"
#include "threadlist.h"
#include "scheduler.h"
#include "ithreadport.h"
#include "quantum.h"
#include "autoalloc.h"
#include "priomap.h"

namespace Mark3
{
class Thread;

//---------------------------------------------------------------------------
using ThreadCreateCallout  = void (*)(Thread* pclThread_);
using ThreadExitCallout    = void (*)(Thread* pclThread_);
using ThreadContextCallout = void (*)(Thread* pclThread_);

//---------------------------------------------------------------------------
class Thread : public TypedLinkListNode<Thread>
{
public:
    void* operator new(size_t sz, void* pv) { return reinterpret_cast<Thread*>(pv); };
    ~Thread();

    Thread() { m_eState = ThreadState::Invalid; }

    bool IsInitialized() { return (m_eState !=
ThreadState::Invalid); }

    void Init(K_WORD*          pwStack_,
              uint16_t         u16StackSize_,
              PORT_PRIO_TYPE   uXPriority_,
              ThreadEntryFunc  pfEntryPoint_,
              void*            pvArg_);

    static Thread*
    Init(uint16_t u16StackSize_, PORT_PRIO_TYPE uXPriority_,
ThreadEntryFunc pfEntryPoint_, void* pvArg_);

    void Start();

    void Stop();

#if KERNEL_NAMED_THREADS

    void SetName(const char* szName_) { m_szName = szName_; }
    const char* GetName() { return m_szName; }
#endif // #if KERNEL_NAMED_THREADS

    ThreadList* GetOwner(void) { return m_pclOwner; }
    inline ThreadList* GetCurrent(void) { return
m_pclCurrent; }
    PORT_PRIO_TYPE GetPriority(void) { return
m_uXPriority; }
    PORT_PRIO_TYPE GetCurPriority(void) { return
m_uXCurPriority; }

#if KERNEL_ROUND_ROBIN

    void SetQuantum(uint16_t u16Quantum_) { m_u16Quantum = u16Quantum_; }
    uint16_t GetQuantum(void) { return m_u16Quantum; }
#endif // #if KERNEL_ROUND_ROBIN

    void SetCurrent(ThreadList* pclNewList_) { m_pclCurrent = pclNewList_;
}
    void SetOwner(ThreadList* pclNewList_) { m_pclOwner = pclNewList_; }
```

```
    void SetPriority(PORT_PRIO_TYPE uXPriority_);

    void InheritPriority(PORT_PRIO_TYPE uXPriority_);

    void Exit();

    static void Sleep(uint32_t u32TimeMs_);

    static void Yield(void);

    static void CoopYield(void);

    void SetID(uint8_t u8ID_) { m_u8ThreadID = u8ID_; }
    uint8_t GetID() { return m_u8ThreadID; }

#if KERNEL_STACK_CHECK

    uint16_t GetStackSlack();
#endif // #if KERNEL_STACK_CHECK

#if KERNEL_EVENT_FLAGS

    uint16_t GetEventFlagMask() { return m_u16FlagMask; }

    void SetEventFlagMask(uint16_t u16Mask_) { m_u16FlagMask = u16Mask_; }

    void SetEventFlagMode(EventFlagOperation eMode_) {
m_eFlagMode = eMode_; }

    EventFlagOperation GetEventFlagMode() { return
m_eFlagMode; }
#endif // #if KERNEL_EVENT_FLAGS

    Timer* GetTimer();

    void SetExpired(bool bExpired_);

    bool GetExpired();

#if KERNEL_EXTENDED_CONTEXT

    void* GetExtendedContext() { return m_pvExtendedContext; }

    void SetExtendedContext(void* pvData_) {
m_pvExtendedContext = pvData_; }
#endif // #if KERNEL_EXTENDED_CONTEXT

    ThreadState GetState() { return m_eState; }
    void SetState(ThreadState eState_) { m_eState = eState_; }

    K_WORD* GetStack() { return m_pwStack; }

    uint16_t GetStackSize() { return m_u16StackSize; }

    friend class ThreadPort;

private:
    static void ContextSwitchSWI(void);

    void SetPriorityBase(PORT_PRIO_TYPE uXPriority_);

    K_WORD* m_pwStackTop;

    K_WORD* m_pwStack;

    uint8_t m_u8ThreadID;

    PORT_PRIO_TYPE m_uXPriority;

    PORT_PRIO_TYPE m_uXCurPriority;

    ThreadState m_eState;

#if KERNEL_EXTENDED_CONTEXT
    void* m_pvExtendedContext;
#endif // #if KERNEL_EXTENDED_CONTEXT

#if KERNEL_NAMED_THREADS
    const char* m_szName;
#endif // #if KERNEL_NAMED_THREADS

    uint16_t m_u16StackSize;

    ThreadList* m_pclCurrent;

    ThreadList* m_pclOwner;

```

```
    ThreadEntryFunc m_pfEntryPoint;

    void* m_pvArg;

#if KERNEL_ROUND_ROBIN
    uint16_t m_u16Quantum;
#endif // #if KERNEL_ROUND_ROBIN

#if KERNEL_EVENT_FLAGS
    uint16_t m_u16FlagMask;

    EventFlagOperation m_eFlagMode;
#endif // #if KERNEL_EVENT_FLAGS

    Timer m_clTimer;

    bool m_bExpired;
};

} // namespace Mark3
```

20.135 /home/moslevin/projects/m3-repo/kernel/src/public/threadlist.h File Reference

Thread linked-list declarations.

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

Classes

- class Mark3::ThreadList

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

Namespaces

- Mark3

20.135.1 Detailed Description

Thread linked-list declarations.

Definition in file threadlist.h.

20.136 threadlist.h

```
/*===========================================================================
     _____     _____     _____     _____
 ___|    _|__ __|_    |__ __|__   |__ __| __  |__  ______
|    \  /  | ||    \      ||     |     ||  |/ /     ||___   |
|     \/   | ||     \     ||     \     ||     \     ||___   |
|__/\__/|__|_||__|\__\  __||__|\__\  __||__|\__\  __||______|
    |_____|      |_____|      |_____|      |_____|

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

Copyright (c) 2012 - 2019 m0slevin, all rights reserved.
See license.txt for more information
=========================================================================== */
#pragma once

#include "kerneltypes.h"
#include "priomap.h"
#include "ll.h"

//---------------------------------------------------------------------------
namespace Mark3
{
class Thread;

class ThreadList : public LinkListNode, public
TypedCircularLinkList<Thread>
{
public:
    void* operator new(size_t sz, void* pv) { return reinterpret_cast<ThreadList*>(pv); };
    ThreadList();

    void SetPriority(PORT_PRIO_TYPE uXPriority_);

    void SetMapPointer(PriorityMap* pclMap_);

    void Add(Thread* node_);

    void Add(Thread* node_, PriorityMap* pclMap_,
PORT_PRIO_TYPE uXPriority_);

    void AddPriority(Thread* node_);

    void Remove(Thread* node_);

    Thread* HighestWaiter();

private:
    PORT_PRIO_TYPE m_uXPriority;

    PriorityMap* m_pclMap;
};
} // namespace Mark3
```

20.137 /home/moslevin/projects/m3-repo/kernel/src/public/threadlistlist.h File Reference

Class implementing a doubly-linked list of thread lists.

`#include "mark3.h"`

Classes

- class Mark3::ThreadListList

    The ThreadListList class Class used to track all threadlists active in the OS kernel. At any point in time, the list can be traversed to get a complete view of all running, blocked, or stopped threads in the system.

Namespaces

- Mark3

20.137.1 Detailed Description

Class implementing a doubly-linked list of thread lists.

Definition in file threadlistlist.h.

20.138 threadlistlist.h

```
/*===========================================================================
     _____        _____        _____        _____
 ___|    _|__  __|_    |__  __|__   |__  __| __  |__  ______
|    \  /  | ||    \      ||     |     ||  |/ /     ||___   |
|     \/   | ||     \     ||     \     ||     \     ||___   |
|__/\__/|__|_||__|\__\  __||__|\__\  __||__|\__\  __||______|
    |_____|      |_____|      |_____|      |_____|

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

Copyright (c) 2012 - 2019 m0slevin, all rights reserved.
See license.txt for more information
=========================================================================== */
#pragma once

#include "mark3.h"

//---------------------------------------------------------------------------
namespace Mark3 {
class ThreadList;

//---------------------------------------------------------------------------
class ThreadListList {
public:
    static void Add(ThreadList* pclThreadList_)
    {
        m_clThreadListList.Add(pclThreadList_);
    }

    static void Remove(ThreadList* pclThreadList_)
    {
        m_clThreadListList.Remove(pclThreadList_);
    }

    static ThreadList* GetHead()
    {
        return m_clThreadListList.GetHead();
    }

private:
    static TypedDoubleLinkList<ThreadList>
 m_clThreadListList;
};
} // namespace Mark3
```

20.139 /home/moslevin/projects/m3-repo/kernel/src/public/timer.h File Reference

Timer object declarations.

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

Classes

- class Mark3::Timer

  The Timer Class. This class provides kernel-managed timers, used to provide high-precision delays. Functionality is useful to both user-code, and is used extensively within the kernel and its blocking objects to implement round-robin scheduling, thread sleep, and timeouts. Provides one-shot and periodic timers for use by application code. This object relies on a target-defined hardware timer implementation, which is multiplexed by the kernel's timer scheduler.

Namespaces

- Mark3

Typedefs

- using Mark3::TimerCallback = void(∗)(Thread ∗pclOwner_, void ∗pvData_)

Variables

- static constexpr auto Mark3::uMaxTimerTicks = uint32_t { 0x7FFFFFFF }
  Maximum value to set.
- static constexpr auto Mark3::uTimerTicksInvalid = uint32_t { 0 }
- static constexpr auto Mark3::uTimerFlagOneShot = uint8_t { 0x01 }
  Timer is one-shot.
- static constexpr auto Mark3::uTimerFlagActive = uint8_t { 0x02 }
  Timer is currently active.
- static constexpr auto Mark3::uTimerFlagCallback = uint8_t { 0x04 }
  Timer is pending a callback.
- static constexpr auto Mark3::uTimerFlagExpired = uint8_t { 0x08 }
  Timer is actually expired.

20.139.1 Detailed Description

Timer object declarations.

Definition in file timer.h.

20.140 timer.h

```
/*===========================================================================
     _____     _____     _____     _____
 ___|    _|__ __|_    |__ __|__   |__ __| __  |__  ______
|    \  /  | ||    \    ||     |    ||  |/ /    ||___   |
|     \/   | ||     \   ||     \    ||  |  \    ||___   |
|__/\__/|__|_||__|\__\  __||__|\__\  __||__|\__\  __||______|
    |_____|      |_____|      |_____|      |_____|

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

Copyright (c) 2012 - 2019 m0slevin, all rights reserved.
See license.txt for more information
=========================================================================== */
#pragma once
#include "kerneltypes.h"
#include "mark3cfg.h"

#include "ll.h"

namespace Mark3
{
class Thread;

//---------------------------------------------------------------------------
static constexpr auto uMaxTimerTicks     = uint32_t { 0x7FFFFFFF };
static constexpr auto uTimerTicksInvalid = uint32_t { 0 };
static constexpr auto uTimerFlagOneShot  = uint8_t { 0x01 };
static constexpr auto uTimerFlagActive   = uint8_t { 0x02 };
static constexpr auto uTimerFlagCallback = uint8_t { 0x04 };
static constexpr auto uTimerFlagExpired  = uint8_t { 0x08 };
```

```
//-----------------------------------------------------------------------
using TimerCallback = void (*)(Thread* pclOwner_, void* pvData_);

//-----------------------------------------------------------------------
class TimerList;
class TimerScheduler;
class Quantum;

//-----------------------------------------------------------------------
class Timer : public TypedLinkListNode<Timer>
{
public:
    void* operator new(size_t sz, void* pv) { return reinterpret_cast<Timer*>(pv); }
    ~Timer() {}

    Timer();

    void Init();

    void Start(bool bRepeat_, uint32_t u32IntervalMs_, TimerCallback pfCallback_, void* pvData_);

    void Start();

    void Stop();

private:
    friend class TimerList;

    void SetInitialized() { m_u8Initialized = m_uTimerInitCookie; }

    bool IsInitialized(void) { return (m_u8Initialized == m_uTimerInitCookie); }

    static inline uint32_t SecondsToTicks(uint32_t x) { return (x) * 1000; }
    static inline uint32_t MSecondsToTicks(uint32_t x) { return (x); }
    static inline uint32_t USecondsToTicks(uint32_t x) { return ((x + 999) / 1000); }

    static constexpr auto m_uTimerInvalidCookie = uint8_t { 0x3C };
    static constexpr auto m_uTimerInitCookie    = uint8_t { 0xC3 };

    uint8_t m_u8Initialized;

    uint8_t m_u8Flags;

    TimerCallback m_pfCallback;

    uint32_t m_u32Interval;

    uint32_t m_u32TimeLeft;

    Thread* m_pclOwner;

    void* m_pvData;
};
} // namespace Mark3
```

20.141 /home/moslevin/projects/m3-repo/kernel/src/public/timerlist.h File Reference

Timer list declarations.

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

Classes

- class Mark3::TimerList

  the TimerList class. This class implements a doubly-linked-list of timer objects.

Namespaces

- Mark3

20.141.1 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.

20.142 timerlist.h

```
/*===========================================================================
     _____        _____        _____        _____
 ___|    _|__  __|_    |__  __|__   |__  __| __  |__  ______
|    \  /  | ||    \      ||     |     ||  |/ /     ||___   |
|     \/   | ||     \     ||     \     ||     \     ||___   |
|__/\__/|__|_||__|\__\  __||__|\__\  __||__|\__\  __||______|
    |_____|      |_____|      |_____|      |_____|

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

Copyright (c) 2012 - 2019 m0slevin, all rights reserved.
See license.txt for more information
=========================================================================== */
#pragma once

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

#include "mutex.h"

namespace Mark3
{
class Timer;

//---------------------------------------------------------------------------
class TimerList : public TypedDoubleLinkList<Timer>
{
public:
    void Init();

    void Add(Timer* pclListNode_);

    void Remove(Timer* pclLinkListNode_);

    void Process();

private:
    uint32_t m_u32NextWakeup;

    bool m_bTimerActive;

    Mutex m_clMutex;
};
} // namespace Mark3
```

20.143 /home/moslevin/projects/m3-repo/kernel/src/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 Mark3::TimerScheduler

  The TimerScheduler Class. This implements a "Static" class used to manage a global list of timers used throughout the system.

Namespaces

- Mark3

20.143.1 Detailed Description

Timer scheduler declarations.

Definition in file timerscheduler.h.

20.144 timerscheduler.h

```
/*===========================================================================
     _____        _____        _____        _____
 ___|    _|__  __|_    |__  __|__   |__  __| __  |__  ______
|    \  /  | ||    \      ||     |     ||  |/ /     ||___   |
|     \/   | ||     \     ||     \     ||     \     ||___   |
|__/\__/|__|_||__|\__\  __||__|\__\  __||__|\__\  __||_______|
    |_____|      |_____|      |_____|      |_____|

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

Copyright (c) 2012 - 2019 m0slevin, all rights reserved.
See license.txt for more information
=========================================================================== */
#pragma once

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

#include "ll.h"
#include "timer.h"
#include "timerlist.h"

namespace Mark3
{
//---------------------------------------------------------------------------
class TimerScheduler
{
public:
    static void Init() { m_clTimerList.Init(); }
    static void Add(Timer* pclListNode_) { m_clTimerList.
Add(pclListNode_); }
    static void Remove(Timer* pclListNode_) { m_clTimerList.
Remove(pclListNode_); }
    static void Process() { m_clTimerList.Process(); }

private:
    static TimerList m_clTimerList;
};
} // namespace Mark3
```

20.145 /home/moslevin/projects/m3-repo/kernel/src/quantum.cpp File Reference

Thread Quantum Implementation for Round-Robin Scheduling.

```
#include "mark3.h"
```

20.145.1 Detailed Description

Thread Quantum Implementation for Round-Robin Scheduling.

Definition in file quantum.cpp.

20.146 quantum.cpp

```
/*===========================================================================
     _____        _____        _____        _____
 ___|    _|__  __|_    |__  __|__    |__  __| __  |__  ______
|    \  /  | ||    \      ||     |      ||  |/ /     ||___   |
|     \/   | ||     \     ||     \      ||    \      ||___   |
|__/\__/|__|_||__|\__\  __||__|\__\  __||__|\__\  __||______|
    |_____|      |_____|      |_____|      |_____|

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

Copyright (c) 2012 - 2019 m0slevin, all rights reserved.
See license.txt for more information
===========================================================================*/
#include "mark3.h"

#if KERNEL_ROUND_ROBIN
namespace Mark3
{
//---------------------------------------------------------------------------
uint16_t Quantum::m_u16TicksRemain;
Thread*  Quantum::m_pclActiveThread;
Thread*  Quantum::m_pclTimerThread;
bool     Quantum::m_bInTimer;

//---------------------------------------------------------------------------
void Quantum::SetInTimer()
{
    const auto cs = CriticalGuard{};
    m_bInTimer = true;

    // Timer is active
    if (m_u16TicksRemain) {
        m_u16TicksRemain--;
    }
}

//---------------------------------------------------------------------------
void Quantum::ClearInTimer()
{
    const auto cs = CriticalGuard{};
    m_bInTimer = false;

    // Timer expired - Pivot the thread list.
    if (m_pclActiveThread && (!m_u16TicksRemain)) {
        auto* pclThreadList = m_pclActiveThread->GetCurrent();
        if (pclThreadList->GetHead() != pclThreadList->GetTail()) {
            pclThreadList->PivotForward();
        }
        m_pclActiveThread = nullptr;
    }
}

//---------------------------------------------------------------------------
void Quantum::Update(Thread* pclTargetThread_)
{
    // Don't cancel the current RR interval if we're being interrupted by
    // the timer thread, or are in the middle of running the timer thread.
    // OR if the thread list only has one thread
    auto* pclThreadList = pclTargetThread_->GetCurrent();
    if ((pclThreadList->GetHead() == pclThreadList->GetTail()) || (pclTargetThread_ ==
m_pclTimerThread)
        || (pclTargetThread_ == m_pclActiveThread) ||
m_bInTimer) {
        return;
    }

    // Update with a new thread and timeout.
    m_pclActiveThread = pclTargetThread_;
    m_u16TicksRemain  = pclTargetThread_->GetQuantum();
}

```

```
//---------------------------------------------------------------------------
void Quantum::Cancel()
{
    m_pclActiveThread = nullptr;
    m_u16TicksRemain  = 0;
}
} // namespace Mark3
#endif // #if KERNEL_ROUND_ROBIN
```

20.147 /home/moslevin/projects/m3-repo/kernel/src/readerwriter.cpp File Reference

Reader-writer lock implementation.

```
#include "mark3.h"
```

Namespaces

- Mark3

20.147.1 Detailed Description

Reader-writer lock implementation.

Definition in file readerwriter.cpp.

20.148 readerwriter.cpp

```
/*===========================================================================
     _____     _____     _____     _____
 ___|    _|__ __|_    |__ __|__   |__ __| __  |__  _____
|    \  /  | ||    \    ||     |    ||  |/ /    ||___   |
|     \/   | ||     \   ||     \    ||     \    ||___   |
|__/\__/|__|_||__|\__\ __||__|\__\ __||__|\__\ __||______|
    |_____|      |_____|      |_____|      |_____|

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

Copyright (c) 2012 - 2019 m0slevin, all rights reserved.
See license.txt for more information
===========================================================================*/
#include "mark3.h"
namespace Mark3
{
//---------------------------------------------------------------------------
void ReaderWriterLock::Init()
{
    m_u8ReadCount = 0;
    m_clGlobalMutex.Init();
    m_clReaderMutex.Init();
}

//---------------------------------------------------------------------------
void ReaderWriterLock::AcquireReader()
{
    AcquireReader_i(0);
}

//---------------------------------------------------------------------------
bool ReaderWriterLock::AcquireReader(uint32_t u32TimeoutMs_)
{
    return AcquireReader_i(u32TimeoutMs_);
}

```

```
//---------------------------------------------------------------------------
void ReaderWriterLock::ReleaseReader()
{
    m_clReaderMutex.Claim();
    m_u8ReadCount--;
    if (0 == m_u8ReadCount) {
        m_clGlobalMutex.Release();
    }
    m_clReaderMutex.Release();
}

//---------------------------------------------------------------------------
void ReaderWriterLock::AcquireWriter()
{
    AcquireWriter_i(0);
}

//---------------------------------------------------------------------------
bool ReaderWriterLock::AcquireWriter(uint32_t u32TimeoutMs_)
{
    return AcquireWriter_i(u32TimeoutMs_);
}

//---------------------------------------------------------------------------
void ReaderWriterLock::ReleaseWriter()
{
    m_clGlobalMutex.Release();
}

//---------------------------------------------------------------------------
bool ReaderWriterLock::AcquireReader_i(uint32_t u32TimeoutMs_)
{
    auto rc = true;
    if (!m_clReaderMutex.Claim(u32TimeoutMs_)) {
        return false;
    }

    m_u8ReadCount++;
    if (1 == m_u8ReadCount) {
        rc = m_clGlobalMutex.Claim(u32TimeoutMs_);
    }

    m_clReaderMutex.Release();
    return rc;
}

//---------------------------------------------------------------------------
bool ReaderWriterLock::AcquireWriter_i(uint32_t u32TimeoutMs_)
{
    return m_clGlobalMutex.Claim(u32TimeoutMs_);
}
} // namespace Mark3
```

20.149 /home/moslevin/projects/m3-repo/kernel/src/scheduler.cpp File Reference

Strict-Priority + Round-Robin thread scheduler implementation.

```
#include "mark3.h"
```

Namespaces

- Mark3

Variables

- Mark3::Thread ∗ g_pclNext
- Mark3::Thread ∗ g_pclCurrent

20.149.1 Detailed Description

Strict-Priority + Round-Robin thread scheduler implementation.

Definition in file scheduler.cpp.

20.149.2 Variable Documentation

20.149.2.1 g_pclCurrent

```
Mark3::Thread* g_pclCurrent
```

Definition at line 25 of file scheduler.cpp.

20.149.2.2 g_pclNext

```
Mark3::Thread* g_pclNext
```

Definition at line 24 of file scheduler.cpp.

20.150 scheduler.cpp

```
00001 /*===========================================================================
00002      _____     _____     _____     _____
00003  ___|    _|__ __|_    |__ __|__   |__ __| __  |__ _____
00004 |    \  /  | ||    \    ||     |     ||  |/ /    ||___   |
00005 |     \/   | ||     \   ||     \     ||     \    ||___   |
00006 |__/\__/|__|_||__|\__\ __||__|\__\ __||__|\__\ __||_____|
00007     |_____|      |_____|      |_____|      |_____|
00008
00009 --[Mark3 Realtime Platform]--------------------------------------------------
00010
00011 Copyright (c) 2012 - 2019 m0slevin, all rights reserved.
00012 See license.txt for more information
00013 ===========================================================================*/
00022 #include "mark3.h"
00023
00024 Mark3::Thread* g_pclNext;
00025 Mark3::Thread* g_pclCurrent;
00026
00027 namespace Mark3
00028 {
00029 bool        Scheduler::m_bEnabled;
00030 bool        Scheduler::m_bQueuedSchedule;
00031 ThreadList  Scheduler::m_clStopList;
00032 ThreadList  Scheduler::m_aclPriorities[
      KERNEL_NUM_PRIORITIES];
00033 PriorityMap Scheduler::m_clPrioMap;
00034
00035 //---------------------------------------------------------------------------
00036 void Scheduler::Init()
00037 {
00038     for (size_t i = 0; i < m_uNumPriorities; i++) {
00039         m_aclPriorities[i].SetPriority(i);
00040         m_aclPriorities[i].SetMapPointer(&
      m_clPrioMap);
00041     }
00042 }
```

```

//---------------------------------------------------------------------------
void Scheduler::Schedule()
{
    auto uXPrio = m_clPrioMap.HighestPriority();
    if (0 == uXPrio) {
        Kernel::Panic(PANIC_NO_READY_THREADS);
    }
    // Priorities are one-indexed
    uXPrio--;

    // Get the thread node at this priority.
    g_pclNext = m_aclPriorities[uXPrio].GetHead();
}

//---------------------------------------------------------------------------
void Scheduler::Add(Thread* pclThread_)
{
    KERNEL_ASSERT(pclThread_ != nullptr);

    m_aclPriorities[pclThread_->GetPriority()].Add(pclThread_);
}

//---------------------------------------------------------------------------
void Scheduler::Remove(Thread* pclThread_)
{
    KERNEL_ASSERT(pclThread_ != nullptr);

    m_aclPriorities[pclThread_->GetPriority()].Remove(pclThread_);
}

//---------------------------------------------------------------------------
bool Scheduler::SetScheduler(bool bEnable_)
{
    const auto cs = CriticalGuard{};
    auto bRet      = m_bEnabled;
    m_bEnabled = bEnable_;
    // If there was a queued scheduler evevent, dequeue and trigger an
    // immediate Yield
    if (m_bEnabled && m_bQueuedSchedule) {
        m_bQueuedSchedule = false;
        Thread::Yield();
    }
    return bRet;
}
} // namespace Mark3
```

20.151 /home/moslevin/projects/m3-repo/kernel/src/thread.cpp File Reference

Platform-Independent thread class Definition.

```
#include "kerneltypes.h"
#include "mark3cfg.h"
#include "mark3.h"
```

Namespaces

- Mark3

20.151.1 Detailed Description

Platform-Independent thread class Definition.

Definition in file thread.cpp.

20.152 thread.cpp

```
/*===========================================================================
     _____        _____        _____        _____
 ___|    _|__  __|_    |__  __|__   |__  __| __  |__  ______
|    \  /  | ||    \      ||     |     ||  |/ /     ||___   |
|     \/   | ||     \     ||     \     ||     \     ||___   |
|__/\__/|__|_||__|\__\  __||__|\__\  __||__|\__\  __||______|
    |_____|      |_____|      |_____|      |_____|

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

Copyright (c) 2012 - 2019 m0slevin, all rights reserved.
See license.txt for more information
===========================================================================*/
#include "kerneltypes.h"
#include "mark3cfg.h"

#include "mark3.h"

namespace Mark3
{
//---------------------------------------------------------------------------
Thread::~Thread()
{
    // On destruction of a thread located on a stack,
    // ensure that the thread is either stopped, or exited.
    // If the thread is stopped, move it to the exit state.
    // If not in the exit state, kernel panic -- it's catastrophic to have
    // running threads on stack suddenly disappear.
    if (ThreadState::Stop == m_eState) {
        const auto cs = CriticalGuard{};
        m_pclCurrent->Remove(this);
        m_pclCurrent = nullptr;
        m_pclOwner   = nullptr;
        m_eState     = ThreadState::Exit;
    } else if (ThreadState::Exit != m_eState) {
        Kernel::Panic(PANIC_RUNNING_THREAD_DESCOPED);
    }
}

//---------------------------------------------------------------------------
void Thread::Init(
    K_WORD* pwStack_, uint16_t u16StackSize_, PORT_PRIO_TYPE uXPriority_, ThreadEntryFunc pfEntryPoint_, void* pvArg_)
{
    static auto u8ThreadID = uint8_t { 0 };

    KERNEL_ASSERT(pwStack_);
    KERNEL_ASSERT(pfEntryPoint_);

    ClearNode();

    m_u8ThreadID = u8ThreadID++;

    // Initialize the thread parameters to their initial values.
    m_pwStack    = pwStack_;
    m_pwStackTop = PORT_TOP_OF_STACK(pwStack_, u16StackSize_);

    m_u16StackSize  = u16StackSize_;
    m_uXPriority    = uXPriority_;
    m_uXCurPriority = m_uXPriority;
    m_pfEntryPoint  = pfEntryPoint_;
    m_pvArg         = pvArg_;

#if KERNEL_NAMED_THREADS
    m_szName = nullptr;
#endif
#if KERNEL_ROUND_ROBIN
    m_u16Quantum = THREAD_QUANTUM_DEFAULT;
#endif

    m_clTimer.Init();

    // Call CPU-specific stack initialization
    ThreadPort::InitStack(this);

    // Add to the global "stop" list.
    { // Begin critical section
        const auto cs = CriticalGuard{};
        m_pclOwner   = Scheduler::GetThreadList(m_uXPriority);
        m_pclCurrent = Scheduler::GetStopList();
        m_eState     = ThreadState::Stop;
        m_pclCurrent->Add(this);
    } // End critical section
```

```
#if KERNEL_THREAD_CREATE_CALLOUT
    ThreadCreateCallout pfCallout = Kernel::GetThreadCreateCallout();
    if (nullptr != pfCallout) {
        pfCallout(this);
    }
#endif
}

//---------------------------------------------------------------------------
Thread* Thread::Init(uint16_t u16StackSize_, PORT_PRIO_TYPE uXPriority_, ThreadEntryFunc pfEntryPoint_, void* pvArg_)
{
    auto* pclNew  = AutoAlloc::NewObject<Thread, AutoAllocType::Thread>();
    auto* pwStack = static_cast<K_WORD*>(AutoAlloc::NewRawData(u16StackSize_));
    pclNew->Init(pwStack, u16StackSize_, uXPriority_, pfEntryPoint_, pvArg_);
    return pclNew;
}

//---------------------------------------------------------------------------
void Thread::Start(void)
{
    KERNEL_ASSERT(IsInitialized());

    // Remove the thread from the scheduler's "stopped" list, and add it
    // to the scheduler's ready list at the proper priority.

    const auto cs = CriticalGuard{};
    Scheduler::GetStopList()->Remove(this);
    Scheduler::Add(this);
    m_pclOwner   = Scheduler::GetThreadList(m_uXPriority);
    m_pclCurrent = m_pclOwner;
    m_eState     = ThreadState::Ready;

#if KERNEL_ROUND_ROBIN
    if (Kernel::IsStarted()) {
        if (GetCurPriority() >= Scheduler::GetCurrentThread()->GetCurPriority()) {
            // Deal with the thread Quantum
            Quantum::Update(this);
        }
    }
#endif

    if (Kernel::IsStarted()) {
        if (GetCurPriority() >= Scheduler::GetCurrentThread()->GetCurPriority()) {
            Thread::Yield();
        }
    }
}

//---------------------------------------------------------------------------
void Thread::Stop()
{
    KERNEL_ASSERT(IsInitialized());

    auto bReschedule = false;
    if (ThreadState::Stop == m_eState) {
        return;
    }

    { // Begin critical section
        const auto cs = CriticalGuard{};

        // If a thread is attempting to stop itself, ensure we call the scheduler
        if (this == Scheduler::GetCurrentThread()) {
            bReschedule = true;
    #if KERNEL_ROUND_ROBIN
            // Cancel RR scheduling
            Quantum::Cancel();
    #endif
        }

        // Add this thread to the stop-list (removing it from active scheduling)
        // Remove the thread from scheduling
        if (ThreadState::Ready == m_eState) {
            Scheduler::Remove(this);
        } else if (ThreadState::Blocked == m_eState) {
            m_pclCurrent->Remove(this);
        }

        m_pclOwner   = Scheduler::GetStopList();
        m_pclCurrent = m_pclOwner;
        m_pclOwner->Add(this);
```

```
        m_eState = ThreadState::Stop;

        // Just to be safe - attempt to remove the thread's timer
        // from the timer-scheduler (does no harm if it isn't
        // in the timer-list)
        TimerScheduler::Remove(&m_clTimer);
    } // End Critical Section

    if (bReschedule) {
        Thread::Yield();
    }
}

//---------------------------------------------------------------------------
void Thread::Exit()
{
    KERNEL_ASSERT(IsInitialized());

    auto bReschedule = false;

    if (ThreadState::Exit == m_eState) {
        return;
    }

    { // Begin critical section
        const auto cs = CriticalGuard{};

        // If this thread is the actively-running thread, make sure we run the
        // scheduler again.
        if (this == Scheduler::GetCurrentThread()) {
            bReschedule = true;
#if KERNEL_ROUND_ROBIN
            // Cancel RR scheduling
            Quantum::Cancel();
#endif
        }

        // Remove the thread from scheduling
        if (ThreadState::Ready == m_eState) {
            Scheduler::Remove(this);
        } else if ((ThreadState::Blocked == m_eState) || (ThreadState::Stop == m_eState)) {
            m_pclCurrent->Remove(this);
        }

        m_pclCurrent = nullptr;
        m_pclOwner   = nullptr;
        m_eState     = ThreadState::Exit;

        // We've removed the thread from scheduling, but interrupts might
        // trigger checks against this thread's currently priority before
        // we get around to scheduling new threads.  As a result, set the
        // priority to idle to ensure that we always wind up scheduling
        // new threads.
        m_uXCurPriority = 0;
        m_uXPriority    = 0;

        // Just to be safe - attempt to remove the thread's timer
        // from the timer-scheduler (does no harm if it isn't
        // in the timer-list)
        TimerScheduler::Remove(&m_clTimer);
    } // End Critical Section

#if KERNEL_THREAD_EXIT_CALLOUT
    ThreadExitCallout pfCallout = Kernel::GetThreadExitCallout();
    if (nullptr != pfCallout) {
        pfCallout(this);
    }
#endif

    if (bReschedule) {
        // Choose a new "next" thread if we must
        Thread::Yield();
    }
}

//---------------------------------------------------------------------------
void Thread::Sleep(uint32_t u32TimeMs_)
{
    auto  clSemaphore    = Semaphore {};
    auto* pclTimer       = g_pclCurrent->GetTimer();
    auto  lTimerCallback = [](Thread* /*pclOwner*/, void* pvData_) {
        auto* pclSemaphore = static_cast<Semaphore*>(pvData_);
        pclSemaphore->Post();
    };

```

```
    // Create a semaphore that this thread will block on
    clSemaphore.Init(0, 1);

    // Create a one-shot timer that will call a callback that posts the
    // semaphore, waking our thread.
    pclTimer->Init();
    pclTimer->Start(false, u32TimeMs_, lTimerCallback, &clSemaphore);

    clSemaphore.Pend();
}

#if KERNEL_STACK_CHECK
//---------------------------------------------------------------------------
uint16_t Thread::GetStackSlack()
{
    KERNEL_ASSERT(IsInitialized());

    auto wBottom = uint16_t { 0 };
    auto wTop    = static_cast<uint16_t>((m_u16StackSize - 1) / sizeof(
K_ADDR));
    auto wMid    = static_cast<uint16_t>(((wTop + wBottom) + 1) / 2);

    { // Begin critical section
        const auto cs = CriticalGuard{};

        // Logarithmic bisection - find the point where the contents of the
        // stack go from 0xFF's to non 0xFF.  Not Definitive, but accurate enough
        while ((wTop - wBottom) > 1) {
#if PORT_STACK_GROWS_DOWN
            if (m_pwStack[wMid] != static_cast<K_WORD>(-1))
#else
            if (m_pwStack[wMid] == static_cast<K_WORD>(-1))
#endif
            {
                wTop = wMid;
            } else {
                wBottom = wMid;
            }
            wMid = (wTop + wBottom + 1) / 2;
        }
    } // End Critical Section

    return wMid * sizeof(K_ADDR);
}
#endif

//---------------------------------------------------------------------------
void Thread::Yield()
{
    const auto cs = CriticalGuard{};
    // Run the scheduler
    if (Scheduler::IsEnabled()) {
        Scheduler::Schedule();

        // Only switch contexts if the new task is different than the old task
        if (g_pclCurrent != g_pclNext) {
#if KERNEL_ROUND_ROBIN
            Quantum::Update(g_pclNext);
#endif
            Thread::ContextSwitchSWI();
        }
    } else {
        Scheduler::QueueScheduler();
    }
}

//---------------------------------------------------------------------------
void Thread::CoopYield(void)
{
    g_pclCurrent->GetCurrent()->PivotForward();
    Yield();
}

//---------------------------------------------------------------------------
void Thread::SetPriorityBase(PORT_PRIO_TYPE /*uXPriority_*/)
{
    KERNEL_ASSERT(IsInitialized());

    GetCurrent()->Remove(this);
    SetCurrent(Scheduler::GetThreadList(
m_uXPriority));
    GetCurrent()->Add(this);
}

//---------------------------------------------------------------------------
void Thread::SetPriority(PORT_PRIO_TYPE uXPriority_)
{
```

```
    KERNEL_ASSERT(IsInitialized());
    auto bSchedule = false;

    { // Begin critical section
        const auto cs = CriticalGuard{};

        // If this is the currently running thread, it's a good idea to reschedule
        // Or, if the new priority is a higher priority than the current thread's.
        if ((this == g_pclCurrent) || (uXPriority_ > g_pclCurrent->GetPriority())) {
            bSchedule = true;
    #if KERNEL_ROUND_ROBIN
            Quantum::Cancel();
    #endif
        }
        Scheduler::Remove(this);

        m_uXCurPriority = uXPriority_;
        m_uXPriority    = uXPriority_;

        Scheduler::Add(this);
    } // End critical section

    if (bSchedule) {
        if (Scheduler::IsEnabled()) {
            { // Begin critical section
                const auto cs = CriticalGuard{};
                Scheduler::Schedule();
#if KERNEL_ROUND_ROBIN
                Quantum::Update(g_pclNext);
#endif
            } // End critical sectin
            Thread::ContextSwitchSWI();
        } else {
            Scheduler::QueueScheduler();
        }
    }
}

//---------------------------------------------------------------------------
void Thread::InheritPriority(PORT_PRIO_TYPE uXPriority_)
{
    KERNEL_ASSERT(IsInitialized());

    SetOwner(Scheduler::GetThreadList(uXPriority_));
    m_uXCurPriority = uXPriority_;
}

//---------------------------------------------------------------------------
void Thread::ContextSwitchSWI()
{
    // Call the context switch interrupt if the scheduler is enabled.
    if (Scheduler::IsEnabled()) {
#if KERNEL_STACK_CHECK
        if (g_pclCurrent && (g_pclCurrent->GetStackSlack() <= Kernel::GetStackGuardThreshold())) {
            Kernel::Panic(PANIC_STACK_SLACK_VIOLATED);
        }
#endif
#if KERNEL_CONTEXT_SWITCH_CALLOUT
        auto pfCallout = Kernel::GetThreadContextSwitchCallout();
        if (nullptr != pfCallout) {
            pfCallout(g_pclCurrent);
        }
#endif
        KernelSWI::Trigger();
    }
}

//---------------------------------------------------------------------------
Timer* Thread::GetTimer()
{
    KERNEL_ASSERT(IsInitialized());
    return &m_clTimer;
}
//---------------------------------------------------------------------------
void Thread::SetExpired(bool bExpired_)
{
    KERNEL_ASSERT(IsInitialized());
    m_bExpired = bExpired_;
}

//---------------------------------------------------------------------------
bool Thread::GetExpired()
{
    KERNEL_ASSERT(IsInitialized());
    return m_bExpired;
```

```
}
} // namespace Mark3
```

20.153 /home/moslevin/projects/m3-repo/kernel/src/threadlist.cpp File Reference

Thread linked-list definitions.

```
#include "mark3.h"
#include "threadlistlist.h"
```

Namespaces

- Mark3

20.153.1 Detailed Description

Thread linked-list definitions.

Definition in file threadlist.cpp.

20.154 threadlist.cpp

```
/*===========================================================================
     _____     _____     _____     _____
 ___|    _|__ __|_    |__ __|__   |__ __| __  |__  ______
|    \  /  | ||    \    ||     |    ||  |/ /    ||___   |
|     \/   | ||     \   ||     \    ||     \    ||___   |
|__/\__/|__|_||__|\__\ __||__|\__\ __||__|\__\ __||______|
    |_____|      |_____|      |_____|      |_____|

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

Copyright (c) 2012 - 2019 m0slevin, all rights reserved.
See license.txt for more information
===========================================================================*/
#include "mark3.h"
#include "threadlistlist.h"
namespace Mark3
{
//---------------------------------------------------------------------------
ThreadList::ThreadList()
    : m_uXPriority(0)
    , m_pclMap(nullptr)
{
}

//---------------------------------------------------------------------------
void ThreadList::SetPriority(PORT_PRIO_TYPE uXPriority_)
{
    m_uXPriority = uXPriority_;
}

//---------------------------------------------------------------------------
void ThreadList::SetMapPointer(PriorityMap* pclMap_)
{
    KERNEL_ASSERT(pclMap_ != nullptr);
    m_pclMap = pclMap_;
}

//---------------------------------------------------------------------------
void ThreadList::Add(Thread* pclThread_)
{
    KERNEL_ASSERT(pclThread_ != nullptr);
```

```

    // If list was empty, add the object for global threadlist tracking
    if (!GetHead()) {
        ThreadListList::Add(this);
    }
    TypedCircularLinkList<Thread>::Add(pclThread_);
    PivotForward();

    // We've specified a bitmap for this threadlist
    if (nullptr != m_pclMap) {
        // Set the flag for this priority level
        m_pclMap->Set(m_uXPriority);
    }
}

//---------------------------------------------------------------------------
void ThreadList::AddPriority(Thread* pclThread_)
{
    KERNEL_ASSERT(node_ != nullptr);
    auto* pclCurr = GetHead();
    if (nullptr == pclCurr) {
        Add(pclThread_);
        return;
    }
    auto uXHeadPri = pclCurr->GetCurPriority();
    auto* pclTail = GetTail();

    // Set the threadlist's priority level, flag pointer, and then add the
    // thread to the threadlist
    auto uXPriority = pclThread_->GetCurPriority();
    do {
        if (uXPriority > pclCurr->GetCurPriority()) {
            break;
        }
        pclCurr = pclCurr->GetNext();
    } while (pclCurr != pclTail);

    // Insert pclNode before pclCurr in the linked list.
    InsertNodeBefore(pclThread_, pclCurr);

    // If the priority is greater than current head, reset
    // the head pointer.
    if (uXPriority > uXHeadPri) {
        SetHead(pclThread_);
        SetTail(GetHead()->GetPrev());
    } else if (pclThread_->GetNext() == GetHead()) {
        SetTail(pclThread_);
    }
}

//---------------------------------------------------------------------------
void ThreadList::Add(Thread* node_, PriorityMap* pclMap_,
PORT_PRIO_TYPE uXPriority_)
{
    // Set the threadlist's priority level, flag pointer, and then add the
    // thread to the threadlist
    SetPriority(uXPriority_);
    SetMapPointer(pclMap_);
    Add(node_);
}

//---------------------------------------------------------------------------
void ThreadList::Remove(Thread* node_)
{
    // Remove the thread from the list
    TypedCircularLinkList<Thread>::Remove(node_);

    // If the list is empty...
    if (nullptr == GetHead()) {
        // No more threads - remove this object from global threadlist tracking
        ThreadListList::Remove(this);
        if (nullptr != m_pclMap) {
            // Clear the bit in the bitmap at this priority level
            m_pclMap->Clear(m_uXPriority);
        }
    }
}

//---------------------------------------------------------------------------
Thread* ThreadList::HighestWaiter()
{
    return GetHead();
}
} // namespace Mark3
```

20.155 /home/moslevin/projects/m3-repo/kernel/src/threadlistlist.cpp File Reference

Class implementing a doubly-linked list of thread lists.

```
#include "mark3.h"
#include "threadlistlist.h"
```

Namespaces

- Mark3

20.155.1 Detailed Description

Class implementing a doubly-linked list of thread lists.

Definition in file threadlistlist.cpp.

20.156 threadlistlist.cpp

```
/*===========================================================================
     _____        _____        _____        _____
 ___|    _|__  __|_    |__  __|__   |__  __| __  |__  ______
|    \  /  | ||    \      ||     |     ||  |/ /     ||___   |
|     \/   | ||     \     ||     \     ||     \     ||___   |
|__/\__/|__|_||__|\__\  __||__|\__\  __||__|\__\  __||______|
    |_____|      |_____|      |_____|      |_____|

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

Copyright (c) 2012 - 2019 m0slevin, all rights reserved.
See license.txt for more information
=========================================================================== */
#include "mark3.h"
#include "threadlistlist.h"

namespace Mark3 {
TypedDoubleLinkList<ThreadList> ThreadListList::m_clThreadListList;
} // namespace Mark3
```

20.157 /home/moslevin/projects/m3-repo/kernel/src/timer.cpp File Reference

Timer implementations.

```
#include "mark3.h"
```

Namespaces

- Mark3

20.157.1 Detailed Description

Timer implementations.

Definition in file timer.cpp.

20.158 timer.cpp

```
/*===========================================================================
     _____     _____     _____     _____
 ___|    _|__ __|_    |__ __|__   |__ __| __  |__  _____
|    \  /  | ||    \      ||     |     ||  |/ /     ||___   |
|     \/   | ||     \     ||     \     ||     \     ||___   |
|__/\__/|__|_||__|\__\  __||__|\__\  __||__|\__\  __||_____|
    |_____|      |_____|      |_____|      |_____|

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

Copyright (c) 2012 - 2019 m0slevin, all rights reserved.
See license.txt for more information
===========================================================================*/
#include "mark3.h"

namespace Mark3
{
TimerList TimerScheduler::m_clTimerList;

//---------------------------------------------------------------------------
Timer::Timer()
{
    m_u8Initialized = m_uTimerInvalidCookie;
    m_u8Flags       = 0;
}

//---------------------------------------------------------------------------
void Timer::Init()
{
    if (IsInitialized()) {
        KERNEL_ASSERT((m_u8Flags & uTimerFlagActive) == 0);
    }

    ClearNode();
    m_u32Interval = 0;
    m_u32TimeLeft = 0;
    m_u8Flags     = 0;

    SetInitialized();
}

//---------------------------------------------------------------------------
void Timer::Start(bool bRepeat_, uint32_t u32IntervalMs_,
TimerCallback pfCallback_, void* pvData_)
{
    KERNEL_ASSERT(IsInitialized());

    if ((m_u8Flags & uTimerFlagActive) != 0) {
        return;
    }

    m_u32Interval = u32IntervalMs_;
    m_pfCallback  = pfCallback_;
    m_pvData      = pvData_;

    if (!bRepeat_) {
        m_u8Flags = uTimerFlagOneShot;
    } else {
        m_u8Flags = 0;
    }

    Start();
}

//---------------------------------------------------------------------------
void Timer::Start()
{
    KERNEL_ASSERT(IsInitialized());

    if ((m_u8Flags & uTimerFlagActive) != 0) {
        return;
```

```
    }

    m_pclOwner = Scheduler::GetCurrentThread();
    TimerScheduler::Add(this);
}

//---------------------------------------------------------------------------
void Timer::Stop()
{
    KERNEL_ASSERT(IsInitialized());
    if ((m_u8Flags & uTimerFlagActive) == 0) {
        return;
    }
    TimerScheduler::Remove(this);
}
} // namespace Mark3
```

20.159 /home/moslevin/projects/m3-repo/kernel/src/timerlist.cpp File Reference

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

`#include "mark3.h"`

Namespaces

- Mark3

20.159.1 Detailed Description

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

Definition in file timerlist.cpp.

20.160 timerlist.cpp

```
/*===========================================================================
     _____     _____     _____     _____
 ___|    _|__ __|_    |__ __|__   |__ __| __  |__  ______
|    \  /  | ||    \    ||     |     ||  |/ /    ||___   |
|     \/   | ||     \   ||     \     ||     \    ||___   |
|__/\__/|__|_||__|\__\  __||__|\__\  __||__|\__\ __||______|
    |_____|      |_____|     |_____|     |_____|

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

Copyright (c) 2012 - 2019 m0slevin, all rights reserved.
See license.txt for more information
===========================================================================*/
#include "mark3.h"
namespace Mark3
{
//---------------------------------------------------------------------------
void TimerList::Init(void)
{
    m_bTimerActive  = false;
    m_u32NextWakeup = 0;
    m_clMutex.Init();
}

//---------------------------------------------------------------------------
void TimerList::Add(Timer* pclListNode_)
{
    KERNEL_ASSERT(nullptr != pclListNode_);
```

```
    auto lock = LockGuard { &m_clMutex };

    pclListNode_->ClearNode();
    TypedDoubleLinkList<Timer>::Add(pclListNode_);

    // Set the initial timer value
    pclListNode_->m_u32TimeLeft = pclListNode_->m_u32Interval;

    // Set the timer as active.
    pclListNode_->m_u8Flags |= uTimerFlagActive;
}

//---------------------------------------------------------------------------
void TimerList::Remove(Timer* pclLinkListNode_)
{
    KERNEL_ASSERT(nullptr != pclLinkListNode_);
    auto lock = LockGuard { &m_clMutex };

    TypedDoubleLinkList<Timer>::Remove(pclLinkListNode_);
    pclLinkListNode_->m_u8Flags &= ~uTimerFlagActive;
}

//---------------------------------------------------------------------------
void TimerList::Process(void)
{
    auto lock = LockGuard { &m_clMutex };

    auto* pclCurr = GetHead();
    // Subtract the elapsed time interval from each active timer.
    while (nullptr != pclCurr) {
        auto* pclNext = pclCurr->GetNext();

        // Active timers only...
        if ((pclCurr->m_u8Flags & uTimerFlagActive) != 0) {
            pclCurr->m_u32TimeLeft--;
            if (0 == pclCurr->m_u32TimeLeft) {
                // Expired -- run the callback. these callbacks must be very fast...
                if (nullptr != pclCurr->m_pfCallback) {
                    pclCurr->m_pfCallback(pclCurr->m_pclOwner, pclCurr->m_pvData);
                }
                if ((pclCurr->m_u8Flags & uTimerFlagOneShot) != 0) {
                    // If this was a one-shot timer, deactivate the timer + remove
                    pclCurr->m_u8Flags |= uTimerFlagExpired;
                    pclCurr->m_u8Flags &= ~uTimerFlagActive;
                    Remove(pclCurr);
                } else {
                    // Reset the interval timer.
                    pclCurr->m_u32TimeLeft = pclCurr->m_u32Interval;
                }
            }
        }
        pclCurr = pclNext;
    }
}

} // namespace Mark3
```

Chapter 21

Example Documentation

21.1 lab10_notifications/main.cpp

This examples demonstrates how to use notifcation objects as a thread synchronization mechanism.

```
/*===========================================================================
     _____     _____     _____     _____
 ___|    _|__ __|_    |__ __|__   |__ __| __  |__  _____
|    \  /  | ||    \     ||     |     ||  |/ /     ||___   |
|     \/   | ||     \    ||     \     ||     \     ||___   |
|__/\__/|__|_||__|\__\  __||__|\__\  __||__|\__\  __||_____|
    |_____|   |_____|   |_____|   |_____|

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

Copyright (c) 2012 - 2019 m0slevin, 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.

===========================================================================*/
extern "C" {
void __cxa_pure_virtual(void) {}
void DebugPrint(const char* szString_);
}

namespace
{
using namespace Mark3;
//---------------------------------------------------------------------------
Thread clApp1Thread;
K_WORD awApp1Stack[PORT_KERNEL_DEFAULT_STACK_SIZE];
void   App1Main(void* unused_);

//---------------------------------------------------------------------------
Thread clApp2Thread;
K_WORD awApp2Stack[PORT_KERNEL_DEFAULT_STACK_SIZE];
void   App2Main(void* unused_);

//---------------------------------------------------------------------------
// idle thread -- do nothing
Thread clIdleThread;
K_WORD awIdleStack[PORT_KERNEL_DEFAULT_STACK_SIZE];
void   IdleMain(void* /*unused_*/)
{
    while (1) {}
```

```
}

//---------------------------------------------------------------------------
// Notification object used in the example.
Notify clNotify;

//---------------------------------------------------------------------------
void App1Main(void* unused_)
{
    while (1) {
        auto bNotified = false;
        // Block the thread until the notification object is signalled from
        // elsewhere.
        clNotify.Wait(&bNotified);

        Kernel::DebugPrint("T1: Notified\n");
    }
}

//---------------------------------------------------------------------------
void App2Main(void* unused_)
{
    while (1) {
        // Wait a while, then signal the notification object

        Kernel::DebugPrint("T2: Wait 1s\n");
        Thread::Sleep(1000);

        Kernel::DebugPrint("T2: Notify\n");
        clNotify.Signal();
    }
}
} // anonymous namespace

using namespace Mark3;
//---------------------------------------------------------------------------
int main(void)
{
    // See the annotations in previous labs for details on init.
    Kernel::Init();
    Kernel::SetDebugPrintFunction(DebugPrint);

    clIdleThread.Init(awIdleStack, sizeof(awIdleStack), 0, IdleMain, 0);
    clIdleThread.Start();

    // 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;
}
```

21.2 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 - 2019 m0slevin, 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.

===========================================================================*/
extern "C" {
void __cxa_pure_virtual(void) {}
void DebugPrint(const char* szString_);
}

namespace
{
using namespace Mark3;
//---------------------------------------------------------------------------
Thread clApp1Thread;
K_WORD awApp1Stack[PORT_KERNEL_DEFAULT_STACK_SIZE];
void   App1Main(void* unused_);

//---------------------------------------------------------------------------
Thread clApp2Thread;
K_WORD awApp2Stack[PORT_KERNEL_DEFAULT_STACK_SIZE];
void   App2Main(void* unused_);

//---------------------------------------------------------------------------
// idle thread -- do nothing
Thread clIdleThread;
K_WORD awIdleStack[PORT_KERNEL_DEFAULT_STACK_SIZE];
void   IdleMain(void* /*unused_*/)
{
    while (1) {}
}

//---------------------------------------------------------------------------
Mailbox clMailbox;
uint8_t au8MBData[100];

typedef struct {
    uint8_t au8Buffer[10];
} MBType_t;

//---------------------------------------------------------------------------
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.

        Kernel::DebugPrint("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);
        }

        Kernel::DebugPrint("Messages End\n");
        Thread::Sleep(2000);
    }
}
} // anonymous namespace

using namespace Mark3;
//---------------------------------------------------------------------------
```

```
int main(void)
{
    // See the annotations in previous labs for details on init.
    Kernel::Init();
    Kernel::SetDebugPrintFunction(DebugPrint);

    clIdleThread.Init(awIdleStack, sizeof(awIdleStack), 0, IdleMain, 0);
    clIdleThread.Start();

    // 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;
}
```

21.3 lab1_kernel_setup/main.cpp

This example demonstrates basic kernel setup with two threads.

```
/*===========================================================================
     _____     _____     _____     _____
 ___|    _|__ __|_    |__ __|__   |__ __| __  |__  ______
|    \  /  | ||    \    ||     |     ||  |/ /     ||___   |
|     \/   | ||     \   ||     \     ||     \     ||___   |
|__/\__/|__|_||__|\__\  __||__|\__\  __||__|\__\ __||______|
    |_____|      |_____|      |_____|      |_____|

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

Copyright (c) 2012 - 2019 m0slevin, 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) {}
void DebugPrint(const char* szString_);
}

namespace
```

```
{
using namespace Mark3;
//---------------------------------------------------------------------------
// 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.
Thread clAppThread;
K_WORD awAppStack[PORT_KERNEL_DEFAULT_STACK_SIZE];
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.
Thread clIdleThread;
K_WORD awIdleStack[PORT_KERNEL_DEFAULT_STACK_SIZE];
void   IdleMain(void* unused_);

//---------------------------------------------------------------------------
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) {
        Kernel::DebugPrint("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.
    }
}
} // anonymous namespace

using namespace Mark3;
//---------------------------------------------------------------------------
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();
    Kernel::SetDebugPrintFunction(DebugPrint);

    // 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;
}
```

21.4 lab2_coroutines/main.cpp

This example demonstrates usage of the coroutine scheduler run from the idle thread.

```
/*===========================================================================
     _____     _____     _____     _____
 ___|    _|__ __|_    |__ __|__   |__ __| __  |__  ______
|    \  /  | ||    \    ||     |     ||  |/ /    ||___   |
|     \/   | ||     \   ||     \     ||     \    ||___   |
|__/\__/|__|_||__|\__\ __||__|\__\ __||__|\__\ __||______|
    |_____|      |_____|     |_____|     |_____|

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

Copyright (c) 2012 - 2019 m0slevin, all rights reserved.
See license.txt for more information
===========================================================================*/
#include "mark3.h"
/*===========================================================================

Lab Example 2:

The following example demonstrates how the Mark3 Coroutine Scheudler can be
used to implement cooperative scheduling in the system.

Cooperatively-scheduled tasks run to completion and are not pre-emptable,
unlike threads.  They can, however be assigned a priority to ensure that
the highest-priority tasks are executed first.  Cooperatively-scheduled
tasks should not block, but may use other Mark3 kernel APIs that cause other
threads to unblock (post semaphores, send messages, etc.).  Note that the
cooperatively-scheduled tasks are also intended to be run from a single
thread context - and so the notion of priority is relative only to the set
of coroutines in the system, and is unrelated to the priority of the threads
in the system.

Lessons covered in this example include:

- Initialize the coroutine scheduler and run the scheduler loop from the
idle thread.
- Activate and run coroutines based on stimulus from outside the thread
running the coroutine scheduler.

Exercise:

- Add a coroutine that is activated by an interrupt
- Add a coroutine that is activated by a timer

Takeaway:

Coroutines are an effective way of providing cooperative multitasking in a
system for tasks that do not require pre-emption, and can benefit from having
relative priorities.

===========================================================================*/
extern "C" {
void __cxa_pure_virtual(void) {}
void DebugPrint(const char* szString_);
}

namespace
{
using namespace Mark3;
//---------------------------------------------------------------------------
// 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.
Thread clAppThread;
K_WORD awAppStack[PORT_KERNEL_DEFAULT_STACK_SIZE / sizeof(
      K_WORD)];
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.
Thread clIdleThread;
K_WORD awIdleStack[PORT_KERNEL_DEFAULT_STACK_SIZE / sizeof(
      K_WORD)];
void   IdleMain(void* unused_);

//---------------------------------------------------------------------------
// Declare 3 co-routines, each with a variable which will be passed into its
// handle function.
Coroutine clCoroutine1;
int iCounter1;

Coroutine clCoroutine2;
int iCounter2;

Coroutine clCoroutine3;
int iCounter3;

//---------------------------------------------------------------------------
// Declare handler functions used by the three coroutines.  In this example,
// the coroutine tasks simply increments a counter and print their identity.
void CoroutineTask1(Coroutine* pclCaller_, void* pvArg_)
{
    Kernel::DebugPrint("Task1\n");
    int* piCounter = static_cast<int*>(pvArg_);
    (*piCounter)++;
}

//---------------------------------------------------------------------------
void CoroutineTask2(Coroutine* pclCaller_, void* pvArg_)
{
    Kernel::DebugPrint("Task2\n");
    int* piCounter = static_cast<int*>(pvArg_);
    (*piCounter)++;
}

//---------------------------------------------------------------------------
void CoroutineTask3(Coroutine* pclCaller_, void* pvArg_)
{
    Kernel::DebugPrint("Task3\n");
    int* piCounter = static_cast<int*>(pvArg_);
    (*piCounter)++;
}

//---------------------------------------------------------------------------
void AppMain(void* unused_)
{
    // In this example, the application task is responsible for activating the
    // various coroutines in a round-robin fashion.
    while (1) {
        Thread::Sleep(100);
        clCoroutine1.Activate();
        Thread::Sleep(100);
        clCoroutine2.Activate();
        Thread::Sleep(100);
        clCoroutine3.Activate();
    }
}

//---------------------------------------------------------------------------
void IdleMain(void* unused_)
{
    while (1) {
        // In this example, use the Idle context to run coroutines as they
        // are scheduled
        auto* pclCoRoutine = CoScheduler::Schedule();
        if (pclCoRoutine != nullptr) {
            pclCoRoutine->Run();
        }
    }
}
} // anonymous namespace

using namespace Mark3;
//---------------------------------------------------------------------------
int main(void)
{
```

```
    // See Lab1 for detailed description of the kernel + initial thread
    // bringup sequence.

    Kernel::Init();
    Kernel::SetDebugPrintFunction(DebugPrint);

    CoScheduler::Init();
    clCoroutine1.Init(1, CoroutineTask1, &iCounter1);
    clCoroutine2.Init(2, CoroutineTask2, &iCounter2);
    clCoroutine3.Init(3, CoroutineTask3, &iCounter3);

    clAppThread.Init(awAppStack, sizeof(awAppStack), 1, AppMain, 0);
    clIdleThread.Init(awIdleStack, sizeof(awIdleStack), 0, IdleMain, 0);

    clAppThread.Start();
    clIdleThread.Start();

    Kernel::Start();

    return 0;
}
```

21.5 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 - 2019 m0slevin, 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.

===========================================================================*/

extern "C" {
void __cxa_pure_virtual(void) {}
void DebugPrint(const char* szString_);
}

namespace
{
using namespace Mark3;
//---------------------------------------------------------------------------
// 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.
Thread clApp1Thread;
K_WORD awApp1Stack[PORT_KERNEL_DEFAULT_STACK_SIZE];
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.
Thread clApp2Thread;
K_WORD awApp2Stack[PORT_KERNEL_DEFAULT_STACK_SIZE];
```

```
void    App2Main(void* unused_);

//---------------------------------------------------------------------------
// idle thread -- do nothing
Thread clIdleThread;
K_WORD awIdleStack[PORT_KERNEL_DEFAULT_STACK_SIZE];
void    IdleMain(void* /*unused_*/)
{
    while (1) {}
}

//---------------------------------------------------------------------------
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;
            Kernel::DebugPrint("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;
            Kernel::DebugPrint("Thread 2 - Did some work\n");
        }
    }
}
} // anonymous namespace

using namespace Mark3;
//---------------------------------------------------------------------------
int main(void)
{
    // See the annotations in lab1.
    Kernel::Init();
    Kernel::SetDebugPrintFunction(DebugPrint);

    clIdleThread.Init(awIdleStack, sizeof(awIdleStack), 0, IdleMain, 0);
    clIdleThread.Start();

    // 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;
}
```

21.6 lab4_semaphores/main.cpp

This example demonstrates how to use semaphores for Thread synchronization.

```
/*===========================================================================
     _____        _____        _____        _____
 ___|    _|__  __|_    |__  __|__   |__  __| __  |__  ______
|    \  /  | ||    \      ||     |     ||  |/ /     ||___   |
|     \/   | ||     \     ||     \     ||     \     ||___   |
|__/\__/|__|_||__|\__\  __||__|\__\  __||__|\__\  __||______|
    |_____|      |_____|      |_____|      |_____|

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

Copyright (c) 2012 - 2019 m0slevin, 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.

===========================================================================*/
extern "C" {
void __cxa_pure_virtual(void) {}
void DebugPrint(const char* szString_);
}

namespace
{
using namespace Mark3;
//---------------------------------------------------------------------------
// 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 (PORT_KERNEL_DEFAULT_STACK_SIZE)
Thread clApp1Thread;
K_WORD awApp1Stack[APP1_STACK_SIZE];
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 (PORT_KERNEL_DEFAULT_STACK_SIZE)
Thread clApp2Thread;
K_WORD awApp2Stack[APP2_STACK_SIZE];
void   App2Main(void* unused_);

//---------------------------------------------------------------------------
// idle thread -- do nothing
Thread clIdleThread;
K_WORD awIdleStack[PORT_KERNEL_DEFAULT_STACK_SIZE];
void   IdleMain(void* /*unused_*/)
{
    while (1) {}
}

//---------------------------------------------------------------------------
// This is the semaphore that we'll use to synchronize two threads in this
// demo application
Semaphore clMySem;

//---------------------------------------------------------------------------
void App1Main(void* unused_)
{
    while (1) {
```

```
        // Wait until the semaphore is posted from the other thread
        Kernel::DebugPrint("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.
        Kernel::DebugPrint("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;
            Kernel::DebugPrint("Posted\n");
            clMySem.Post();
        }
    }
}
} // anonymous namespace

using namespace Mark3;
//---------------------------------------------------------------------------
int main(void)
{
    // See the annotations in previous labs for details on init.
    Kernel::Init();
    Kernel::SetDebugPrintFunction(DebugPrint);

    clIdleThread.Init(awIdleStack, sizeof(awIdleStack), 0, IdleMain, 0);
    clIdleThread.Start();

    // 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;
}
```

21.7 lab5_mutexes/main.cpp

This example demonstrates how to use mutexes to protect against concurrent access to resources.

```
/*============================================================================
     _____     _____     _____     _____
 ___|    _|__ __|_    |__ __|__   |__ __| __  |__  ______
|    \  /  | ||    \     ||     |     ||  |/ /     ||___   |
|     \/   | ||     \    ||     \     ||     \     ||___   |
|__/\__/|__|_||_____\  __||__|\__\  __||__|\__\  __||______|
    |_____|      |_____|      |_____|      |_____|
```

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

Copyright (c) 2012 - 2019 m0slevin, 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:

===========================================================================*/
extern "C" {
void __cxa_pure_virtual(void) {}
void DebugPrint(const char* szString_);
}
namespace
{
using namespace Mark3;

//---------------------------------------------------------------------------
// 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 (PORT_KERNEL_DEFAULT_STACK_SIZE)
Thread clApp1Thread;
K_WORD awApp1Stack[APP1_STACK_SIZE];
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 (PORT_KERNEL_DEFAULT_STACK_SIZE)
Thread clApp2Thread;
K_WORD awApp2Stack[APP2_STACK_SIZE];
void   App2Main(void* unused_);

//---------------------------------------------------------------------------
// idle thread -- do nothing
Thread clIdleThread;
K_WORD awIdleStack[PORT_KERNEL_DEFAULT_STACK_SIZE];
void   IdleMain(void* /*unused_*/)
{
    while (1) {}
}

//---------------------------------------------------------------------------
// This is the mutex that we'll use to synchronize two threads in this
// demo application.
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.
volatile uint32_t u32Counter = 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.

        Kernel::DebugPrint("Thread1: Start\n");
        u32Counter++;
        while (u32Counter <= 1000000) { u32Counter++; }
        u32Counter = 0;
        Kernel::DebugPrint("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.

        Kernel::DebugPrint("Thread2: Start\n");
        u32Counter++;
        while (u32Counter <= 1000000) { u32Counter++; }
        u32Counter = 0;
        Kernel::DebugPrint("Thread2: Done\n");

        // Release the lock, allowing the other thread to do its thing.
        clMyMutex.Release();
    }
}
} // anonymous namespace

using namespace Mark3;

//---------------------------------------------------------------------------
int main(void)
{
    // See the annotations in previous labs for details on init.
    Kernel::Init();
    Kernel::SetDebugPrintFunction(DebugPrint);

    clIdleThread.Init(awIdleStack, sizeof(awIdleStack), 0, IdleMain, 0);
    clIdleThread.Start();

    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;
}
```

21.8 lab6_timers/main.cpp

This example demonstrates how to create and use software timers.

```
/*===========================================================================
     _____     _____     _____     _____
 ___|    _|__ __|_    |__ __|__   |__ __| __  |__  ______
|    \  /  | ||    \    ||     |     ||  |/ /     ||___   |
|     \/   | ||     \   ||     \     ||     \     ||___   |
|__/\__/|__|_||__|\__\  __||__|\__\  __||__|\__\  __||______|
    |_____|      |_____|      |_____|      |_____|

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

Copyright (c) 2012 - 2019 m0slevin, 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:
```

```
Demonstration of the periodic and one-shot timer APIs provided by Mark3

Takeaway:

Mark3 can be used to provide flexible one-shot and periodic timers.

===========================================================================*/
extern "C" {
void __cxa_pure_virtual(void) {}
void DebugPrint(const char* szString_);
}

namespace
{
using namespace Mark3;
//---------------------------------------------------------------------------
// 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 (PORT_KERNEL_DEFAULT_STACK_SIZE)
Thread clApp1Thread;
K_WORD awApp1Stack[APP1_STACK_SIZE];
void   App1Main(void* unused_);

//---------------------------------------------------------------------------
// idle thread -- do nothing
Thread clIdleThread;
K_WORD awIdleStack[PORT_KERNEL_DEFAULT_STACK_SIZE];
void   IdleMain(void* /*unused_*/)
{
    while (1) {}
}

//---------------------------------------------------------------------------
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.
    auto* pclSem = static_cast<Semaphore*>(pvData_);
    pclSem->Post();
}

//---------------------------------------------------------------------------
void OneShotCallback(Thread* owner, void* pvData_)
{
    Kernel::DebugPrint("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.
        Kernel::DebugPrint("Thread Triggered.\n");
    }
}
} // anonymous namespace

using namespace Mark3;
```

```
//---------------------------------------------------------------------------
int main(void)
{
    // See the annotations in previous labs for details on init.
    Kernel::Init();
    Kernel::SetDebugPrintFunction(DebugPrint);

    clIdleThread.Init(awIdleStack, sizeof(awIdleStack), 0, IdleMain, 0);
    clIdleThread.Start();

    clApp1Thread.Init(awApp1Stack, sizeof(awApp1Stack), 1, App1Main, 0);

    clApp1Thread.Start();

    Kernel::Start();

    return 0;
}
```

21.9 lab7_events/main.cpp

This example demonstrates how to create and use event groups

```
/*===========================================================================
     _____     _____     _____     _____
 ___|    _|__ __|_    |__ __|__   |__ __| __  |__ _____
|    \  /  | ||    \     ||     |     ||  |/ /     ||___   |
|     \/   | ||     \    ||     \     ||     \     ||___   |
|__/\__/|__|_||__|\__\  __||__|\__\  __||__|\__\  __||_____|
    |_____|      |_____|      |_____|      |_____|

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

Copyright (c) 2012 - 2019 m0slevin, 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 EventFlagOperation::Any_Set and EventFlagOperation::All_Set, 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.

===========================================================================*/

extern "C" {
void __cxa_pure_virtual(void) {}
void DebugPrint(const char* szString_);
}

namespace
{
using namespace Mark3;

//---------------------------------------------------------------------------
// 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 (PORT_KERNEL_DEFAULT_STACK_SIZE)
Thread clApp1Thread;
K_WORD awApp1Stack[APP1_STACK_SIZE];
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 (PORT_KERNEL_DEFAULT_STACK_SIZE)
```

```
Thread clApp2Thread;
K_WORD awApp2Stack[APP2_STACK_SIZE];
void   App2Main(void* unused_);

//---------------------------------------------------------------------------
// idle thread -- do nothing
Thread clIdleThread;
K_WORD awIdleStack[PORT_KERNEL_DEFAULT_STACK_SIZE];
void   IdleMain(void* /*unused_*/)
{
    while (1) {}
}

//---------------------------------------------------------------------------
EventFlag clFlags;

//---------------------------------------------------------------------------
void App1Main(void* unused_)
{
    while (1) {
        // 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 EventFlagOperation::Any_Set 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 EventFlagOperation::All_Set specified.
        auto u16Flags = clFlags.Wait(0xFFFF, EventFlagOperation::Any_Set);

        // Print a message indicaating which bit was set this time.
        switch (u16Flags) {
            case 0x0001: Kernel::DebugPrint("Event1\n"); break;
            case 0x0002: Kernel::DebugPrint("Event2\n"); break;
            case 0x0004: Kernel::DebugPrint("Event3\n"); break;
            case 0x0008: Kernel::DebugPrint("Event4\n"); break;
            case 0x0010: Kernel::DebugPrint("Event5\n"); break;
            case 0x0020: Kernel::DebugPrint("Event6\n"); break;
            case 0x0040: Kernel::DebugPrint("Event7\n"); break;
            case 0x0080: Kernel::DebugPrint("Event8\n"); break;
            case 0x0100: Kernel::DebugPrint("Event9\n"); break;
            case 0x0200: Kernel::DebugPrint("Event10\n"); break;
            case 0x0400: Kernel::DebugPrint("Event11\n"); break;
            case 0x0800: Kernel::DebugPrint("Event12\n"); break;
            case 0x1000: Kernel::DebugPrint("Event13\n"); break;
            case 0x2000: Kernel::DebugPrint("Event14\n"); break;
            case 0x4000: Kernel::DebugPrint("Event15\n"); break;
            case 0x8000: Kernel::DebugPrint("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;
        }
    }
}
} // anonymous namespace

using namespace Mark3;
```

```
//---------------------------------------------------------------------------
int main(void)
{
    // See the annotations in previous labs for details on init.
    Kernel::Init();
    Kernel::SetDebugPrintFunction(DebugPrint);

    clIdleThread.Init(awIdleStack, sizeof(awIdleStack), 0, IdleMain, 0);
    clIdleThread.Start();

    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;
}
```

21.10 lab8_messages/main.cpp

This example demonstrates how to pass data between threads using message passing.

```
/*===========================================================================
     _____     _____     _____     _____
 ___|    _|__ __|_    |__ __|__   |__ __| __  |__  _____
|    \  /  | ||    \      ||     |     ||  |/ /     ||___   |
|     \/   | ||     \     ||     \     ||     \     ||___   |
|__/\__/|__|_||__|\__\  __||__|\__\  __||__|\__\  __||______|
    |_____|      |_____|      |_____|      |_____|

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

Copyright (c) 2012 - 2019 m0slevin, 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.

===========================================================================*/
extern "C" {
void __cxa_pure_virtual(void) {}
void DebugPrint(const char* szString_);
}

namespace
{
using namespace Mark3;
//---------------------------------------------------------------------------
// 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 (PORT_KERNEL_DEFAULT_STACK_SIZE)
Thread clApp1Thread;
K_WORD awApp1Stack[APP1_STACK_SIZE];
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 (PORT_KERNEL_DEFAULT_STACK_SIZE)
Thread clApp2Thread;
K_WORD awApp2Stack[APP2_STACK_SIZE];
void   App2Main(void* unused_);

//---------------------------------------------------------------------------
// idle thread -- do nothing
Thread clIdleThread;
K_WORD awIdleStack[PORT_KERNEL_DEFAULT_STACK_SIZE];
void   IdleMain(void* /*unused_*/)
{
    while (1) {}
}

//---------------------------------------------------------------------------
MessageQueue clMsgQ;

#define MESSAGE_POOL_SIZE (3)
MessagePool s_clMessagePool;
Message     s_clMessages[MESSAGE_POOL_SIZE];

//---------------------------------------------------------------------------
void App1Main(void* unused_)
{
    auto 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
        auto* pclMsg = s_clMessagePool.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.
        auto* pclMsg = clMsgQ.Receive();

        // We received a message, now print out its information
        Kernel::DebugPrint("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.
        s_clMessagePool.Push(pclMsg);
    }
}
} // anonymous namespace

using namespace Mark3;
//---------------------------------------------------------------------------
int main(void)
{
    // See the annotations in previous labs for details on init.
    Kernel::Init();
    Kernel::SetDebugPrintFunction(DebugPrint);
```

```
    clIdleThread.Init(awIdleStack, sizeof(awIdleStack), 0, IdleMain, 0);
    clIdleThread.Start();

    clApp1Thread.Init(awApp1Stack, sizeof(awApp1Stack), 1, App1Main, 0);
    clApp2Thread.Init(awApp2Stack, sizeof(awApp2Stack), 1, App2Main, 0);

    clApp1Thread.Start();
    clApp2Thread.Start();

    clMsgQ.Init();

    s_clMessagePool.Init();
    for (int i = 0; i < MESSAGE_POOL_SIZE; i++) {
        s_clMessages[i].Init();
        s_clMessagePool.Push(&s_clMessages[i]);
    }

    Kernel::Start();

    return 0;
}
```

21.11 lab9_dynamic_threads/main.cpp

This example demonstrates how to create and destroy threads dynamically at runtime.

```
/*===========================================================================
     _____     _____     _____     _____
 ___|    _|__ __|_    |__ __|__   |__ __| __  |__  ______
|    \  /  | ||    \      ||     |     ||  |/ /     ||___   |
|     \/   | ||     \     ||     \     ||     \     ||___   |
|__/\__/|__|_||__|\__\  __||__|\__\  __||__|\__\  __||______|
    |_____|      |_____|      |_____|      |_____|

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

Copyright (c) 2012 - 2019 m0slevin, 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.

===========================================================================*/

extern "C" {
void __cxa_pure_virtual(void) {}
void DebugPrint(const char* szString_);
}

namespace
{
using namespace Mark3;

//---------------------------------------------------------------------------
// 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.
Thread clApp1Thread;
K_WORD awApp1Stack[PORT_KERNEL_DEFAULT_STACK_SIZE];
void   App1Main(void* unused_);

//---------------------------------------------------------------------------
// This block declares the thread stack data for a thread that we'll create
// dynamically.
```

```
K_WORD awApp2Stack[PORT_KERNEL_DEFAULT_STACK_SIZE];

//---------------------------------------------------------------------------
// idle thread -- do nothing
Thread clIdleThread;
K_WORD awIdleStack[PORT_KERNEL_DEFAULT_STACK_SIZE];
void   IdleMain(void* /*unused_*/)
{
    while (1) {}
}

#define MAX_THREADS (10)
Thread*  apclActiveThreads[10];
uint32_t au32ActiveTime[10];

void PrintThreadSlack(void)
{
    Kernel::DebugPrint("Stack Slack");
    for (uint8_t i = 0; i < MAX_THREADS; i++) {
        if (apclActiveThreads[i] != 0) {
            char szStr[10];

            auto u16Slack = apclActiveThreads[i]->GetStackSlack();
            MemUtil::DecimalToHex((K_ADDR)apclActiveThreads[i], szStr);
            Kernel::DebugPrint(szStr);
            Kernel::DebugPrint(" ");
            MemUtil::DecimalToString(u16Slack, szStr);
            Kernel::DebugPrint(szStr);
            Kernel::DebugPrint("\n");
        }
    }
}

void PrintCPUUsage(void)
{
    Kernel::DebugPrint("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]);
        }
    }
}

void ThreadCreate(Thread* pclThread_)
{
    Kernel::DebugPrint("TC\n");
    CriticalSection::Enter();
    for (uint8_t i = 0; i < MAX_THREADS; i++) {
        if (apclActiveThreads[i] == 0) {
            apclActiveThreads[i] = pclThread_;
            break;
        }
    }
    CriticalSection::Exit();

    PrintThreadSlack();
    PrintCPUUsage();
}

void ThreadExit(Thread* pclThread_)
{
    Kernel::DebugPrint("TX\n");
    CriticalSection::Enter();
    for (uint8_t i = 0; i < MAX_THREADS; i++) {
        if (apclActiveThreads[i] == pclThread_) {
            apclActiveThreads[i] = 0;
            au32ActiveTime[i]    = 0;
            break;
        }
    }
    CriticalSection::Exit();

    PrintThreadSlack();
    PrintCPUUsage();
}

void ThreadContextSwitch(Thread* pclThread_)
{
    Kernel::DebugPrint("CS\n");
    static uint32_t u32LastTicks = 0;
    auto            u32Ticks     = Kernel::GetTicks();

    CriticalSection::Enter();
    for (uint8_t i = 0; i < MAX_THREADS; i++) {
        if (apclActiveThreads[i] == pclThread_) {
            au32ActiveTime[i] += u32Ticks - u32LastTicks;
            break;
```

```
        }
    }
    CriticalSection::Exit();

    u32LastTicks = u32Ticks;
}

//---------------------------------------------------------------------------
void WorkerMain1(void* arg_)
{
    auto*    pclSem   = static_cast<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++; }

    Kernel::DebugPrint("Worker1 -- Done Work\n");
    pclSem->Post();

    // Work is completed, just spin now.  Let another thread destory u16.
    while (1) {}
}
//---------------------------------------------------------------------------
void WorkerMain2(void* arg_)
{
    uint32_t u32Count = 0;
    while (u32Count < 1000000) { u32Count++; }

    Kernel::DebugPrint("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++; }

        Kernel::DebugPrint("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++; }

        Kernel::DebugPrint("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() != ThreadState::Exit) {}

        Kernel::DebugPrint("  Test Done\n");
        Thread::Sleep(1000);
        PrintThreadSlack();
    }
}
} // anonymous namespace

using namespace Mark3;
//---------------------------------------------------------------------------
int main(void)
```

```
{
    // See the annotations in previous labs for details on init.
    Kernel::Init();
    Kernel::SetDebugPrintFunction(DebugPrint);

    Kernel::SetThreadCreateCallout(ThreadCreate);
    Kernel::SetThreadExitCallout(ThreadExit);
    Kernel::SetThreadContextSwitchCallout(ThreadContextSwitch);

    clIdleThread.Init(awIdleStack, sizeof(awIdleStack), 0, IdleMain, 0);
    clIdleThread.Start();

    clApp1Thread.Init(awApp1Stack, sizeof(awApp1Stack), 1, App1Main, 0);
    clApp1Thread.Start();
    Kernel::Start();

    return 0;
}
```

Index