atomkernel.c File Reference

#include <stddef.h>
#include "atom.h"

Defines

#define STACK_CHECK_BYTE 0x5A

Functions

Void AtomSched (uint8_t timer_tick)
uint8_t AtomThreadCreate (ATOM_TCB *tcb_ptr, uint8_t priority, void(*entry_point) (uint32_t), uint32_t entry_param, void *stack_top, uint32_t stack_size)
Void AtomIntEnter (void)
Void atomIntExit (uint8_t timer_tick)
ATOM_TCB * atomCurrentContext (void)
uint8_t AtomOSInit (void *idle_thread_stack_top, uint32_t idle_thread_stack_size)
void atomOSStart (void)
uint8_t tcbEnqueuePriority (ATOM_TCB **tcb_queue_ptr, ATOM_TCB *tcb_ptr)
ATOM_TCB * tcbDequeueHead (ATOM_TCB **tcb_queue_ptr)
ATOM_TCB * tcbDequeueEntry (ATOM_TCB **tcb_queue_ptr, ATOM_TCB *tcb_ptr)
ATOM_TCB * tcbDequeuePriority (ATOM_TCB **tcb_queue_ptr, uint8_t priority)

Variables

ATOM_TCB * tcbReadyQ = NULL
uint8_t atomOSStarted = FALSE

Detailed Description

Kernel library.

This module implements the core kernel functionality of managing threads, context-switching and interrupt handlers. It also contains functions for managing queues of TCBs (task control blocks) which are used not only for the queue of ready threads, but also by other OS primitives (such as semaphores) for generically managing lists of TCBs.

Core kernel functionality such as managing the queue of ready threads and how context-switch decisions are made is described within the code. However a quick summary is as follows:

There is a ready queue of threads. There must always be at least one thread ready-to-run. If no application threads are ready, the internal kernel idle thread will be run. This ensures that there is a thread to run at all times.

Application code creates threads using ***atomThreadCreate()***. These threads are added to the ready queue and eventually run when it is their turn (based on priority). When threads are currently-running they are taken off the ready queue. Threads continue to run until:

- They schedule themselves out by calling an OS primitive which blocks, such as a timer delay or blocking on a semaphore. At this point they are placed on the queue of the OS primitive in which they are blocking (for example a timer delay or semaphore).
- They are preempted by a higher priority thread. This could happen at any time if a kernel call from the currently-running thread or from an interrupt handler makes a higher priority thread ready-to-run. Generally this will occur immediately, and while the previously-running thread is still considered ready-to-run, it is no longer the currently-running thread so goes back on to the ready queue.
- They are scheduled out after a timeslice when another thread of the same priority is also ready. This happens on a timer tick, and ensures that threads of the same priority share timeslices. In this case the previously-running thread is still considered ready-to-run so is placed back on to the ready queue.

Thread scheduling decisions are made by ***atomSched()***. This is called at several times, but should never be called by application code directly:

- **After interrupt handlers:** The scheduler is called after every interrupt handler has completed. This allows for any threads which have been made ready-to-run by the interrupt handler to be scheduled in. For example if an interrupt handler posts a semaphore which wakes up a thread of higher priority than the currently-running thread, then the end of interrupt handler reschedule will schedule that thread in.
- ***On timer ticks:*** The timer tick is implemented as an interrupt handler so the end of interrupt call to the scheduler is made as normal, except that in this case round-robin rescheduling is allowed (where threads of the same priority are given a timeslice each in round-robin fashion). This must only occur on timer ticks when the system tick count is incremented.
- **After any OS call changes ready states:** Any OS primitives which change the running state of a thread will call the scheduler to ensure that the change of thread state is noted. For example if a new thread is created using **atomThreadCreate()**, it will internally call the scheduler in case the newly-created thread is higher priority than the currently-running thread. Similarly OS primitives such as semaphores often make changes to a thread's running state. If a thread is going to sleep blocking on a semaphore then the scheduler will be run to ensure that some other thread is scheduled in its place. If a thread is woken by a semaphore post, the scheduler will also be called in case that thread should now be scheduled in (note that when semaphores are posted from an interrupt handler this is deferred to the end of interrupt scheduler call).

When a thread reschedule needs to take place, the scheduler calls out to the architecture-specific port to perform the context-switch, using ***archContextSwitch()*** which must be provided by each architecture port. This function carries out the low-level saving and restoring of registers appropriate for the architecture. The thread being switched out must have a set of CPU registers saved, and the thread being scheduled in has a set of CPU registers restored (which were previously saved). In this fashion threads are rescheduled with the CPU registers in exactly the same state as when the thread was scheduled out.

New threads which have never been scheduled in have a pre-formatted stack area containing a set of CPU register values ready for restoring that appears exactly as if the thread had been previously scheduled out. In other words, the scheduler need not know when it restores registers to switch a thread in whether it has previously run or

if it has never been run since the thread was created. The context-save area is formatted in exactly the same manner.

Functions contained in this module:

Application-callable initialisation functions:

- atomOSInit(): Initialises the operating system.
- atomOSStart(): Starts the OS running (with the highest priority thread).

Application-callable general functions:

- atomThreadCreate(): Thread creation API.
- atomCurrentContext(): Used by kernel and application code to check whether the thread is currently running at thread or interrupt context. This is very useful for implementing safety checks and preventing interrupt handlers from making kernel calls that would block.
- **AtomIntEnter()** / **atomIntExit()**: Must be called by any interrupt handlers.

Internal kernel functions:

- atomSched(): Core scheduler.
- atomThreadSwitch(): Context-switch routine.
- atomIdleThread(): Simple thread to be run when no other threads ready.
- tcbEnqueuePriority(): Enqueues TCBs (task control blocks) on lists.
- tcbDequeueHead(): Dequeues the head of a TCB list.
- tcbDequeueEntry(): Dequeues a particular entry from a TCB list.
- tcbDequeuePriority(): Dequeues an entry from a TCB list using priority.

Define Documentation

```
#define STACK_CHECK_BYTE   0x5A
```

Bytecode to fill thread stacks with for stack-checking purposes

Referenced by atomThreadCreate().

atomCurrentContext

```
ATOM_TCB* atomCurrentContext  (  void      )
```

Get the current thread context. Returns a pointer to the current thread's TCB, or NULL if not in thread-context (in interrupt context).

Return values:

Pointer to current thread's TCB, NULL if in interrupt context

Referenced by atomMutexDelete(), atomMutexGet(), atomMutexPut(), atomQueueDelete(), atomQueueGet(), atomQueuePut(), atomSemDelete(), atomSemGet(), atomSemPut(), atomThreadCreate(), and atomTimerDelay().

atomIntEnter

```
void atomIntEnter ( void )
```

Interrupt handler entry routine.

Must be called at the start of any interrupt handlers that may call an OS primitive and make a thread ready.

Returns:

None

atomIntExit

```
void atomIntExit ( uint8_t timer_tick )
```

Interrupt handler exit routine.

Must be called at the end of any interrupt handlers that may call an OS primitive and make a thread ready.

This is responsible for calling the scheduler at the end of interrupt handlers to determine whether a new thread has now been made ready and should be scheduled in.

Parameters:

timer_tick TRUE if this is a timer tick

Returns:

None

References atomSched().

atomOSInit

```
uint8_t atomOSInit ( void *     idle_thread_stack_top,
                     uint32_t   idle_thread_stack_size
                   )
```

Initialise the atomthreads OS.

Must be called before any application code uses the atomthreads APIs. No threads are actually started until the application calls atomOSStart().

Callers must provide a pointer to some storage for the idle thread stack. The caller is responsible for calculating the appropriate space required for their particular architecture.

Applications should use the following initialisation sequence:

- Call atomOSInit() before calling any atomthreads APIs
- Arrange for a timer to call atomTimerTick() periodically
- Create one or more application threads using atomThreadCreate()
- Start the OS using atomOSStart(). At this point the highest priority application thread created will be started.

Interrupts should be disabled until the first thread restore is complete, to avoid any complications due to interrupts occurring while crucial operating system facilities are being initialised. They are normally enabled by the archFirstThreadRestore() routine in the architecture port.

Parameters:

[in]	*idle_thread_stack_top*	Ptr to top of stack area for idle thread
[in]	*idle_thread_stack_size*	Size of idle thread stack in bytes

Return values:

ATOM_OK	Success
ATOM_ERROR	Initialisation error

References atomOSStarted, atomThreadCreate(), FALSE, IDLE_THREAD_PRIORITY, and uint8_t.

atomOSStart

void atomOSStart (void)

Start the highest priority thread running.

This function must be called after all OS initialisation is complete, and at least one application thread has been created. It will start executing the highest priority thread created (or first created if multiple threads share the highest priority).

Interrupts must still be disabled at this point. They must only be enabled when the first thread is restored and started by the architecture port's archFirstThreadRestore() routine.

Returns:

None

Enable the OS started flag. This stops routines like atomThreadCreate() attempting to schedule in a newly-created thread until the scheduler is up and running.

Application calls to atomThreadCreate() should have added at least one thread to the ready queue. Take the highest priority one off and schedule it in. If no threads were created, the OS will simply start the idle thread (the lowest priority allowed to be scheduled is the idle thread's priority, 255).

References archFirstThreadRestore(), atomOSStarted, tcbDequeuePriority(), and TRUE.

atomSched

void atomSched (uint8_t *timer_tick*)

This is an internal function not for use by application code.

This is the main scheduler routine. It is called by the various OS library routines to check if any threads should be scheduled in now. If so, the context will be switched from the current thread to the new one.

The scheduler is priority-based with round-robin performed on threads with the same priority. Round-robin is only performed on timer ticks however. During reschedules caused by an OS operation (e.g. after giving or taking a semaphore) we only allow the scheduling in of threads with higher priority than current priority. On timer ticks we also allow the scheduling of same-priority threads - in that case we schedule in the head of the ready list for that priority and put the current thread at the tail.

Parameters:

`[in]` *timer_tick* Should be TRUE when called from the system tick

Returns:

None

Check the OS has actually started. As long as the proper initialisation sequence is followed there should be no calls here until the OS is started, but we check to handle badly-behaved ports.

If the current thread is going into suspension, then unconditionally dequeue the next thread for execution.

Dequeue the next ready to run thread. There will always be at least the idle thread waiting. Note that this could actually be the suspending thread if it was unsuspended before the scheduler was called.

Don't need to add the current thread to any queue because it was suspended by another OS mechanism and will be sitting on a suspend queue or similar within one of the OS primitive libraries (e.g. semaphore).

Otherwise the current thread is still ready, but check if any other threads are ready.

Current priority is already highest (0), don't allow preempt by threads of any priority because this is not a time-slice.

References atomOSStarted, CRITICAL_END, CRITICAL_START, CRITICAL_STORE, FALSE, int16_t, atom_tcb::priority, atom_tcb::suspended, tcbDequeueHead(), tcbDequeuePriority(), tcbEnqueuePriority(), TRUE, and uint8_t.

Referenced by atomIntExit(), atomMutexDelete(), atomMutexGet(), atomMutexPut(), atomQueueDelete(), atomQueueGet(), atomQueuePut(), atomSemDelete(), atomSemGet(), atomSemPut(), atomThreadCreate(), and atomTimerDelay().

atomThreadCreate

```
uint8_t atomThreadCreate (  ATOM_TCB *          tcb_ptr,
                            uint8_t             priority,
                            void(*)(uint32_t)   entry_point,
                            uint32_t            entry_param,
                            void *              stack_top,
                            uint32_t            stack_size
                          )
```

Creates and starts a new thread.

Callers provide the ATOM_TCB structure storage, these are not obtained from an internal TCB free list.

The function puts the new thread on the ready queue and calls the scheduler. If the priority is higher than the current priority, then the new thread may be scheduled in before the function returns.

Optionally prefills the thread stack with a known value to enable stack usage checking (if the ATOM_STACK_CHECKING macro is defined).

Parameters:

[in]	*tcb_ptr*	Pointer to the thread's TCB storage
[in]	*priority*	Priority of the thread (0 to 255)
[in]	*entry_point*	Thread entry point
[in]	*entry_param*	Parameter passed to thread entry point
[in]	*stack_top*	Top of the stack area
[in]	*stack_size*	Size of the stack area in bytes

Return values:

ATOM_OK	Success
ATOM_ERR_PARAM	Bad parameters
ATOM_ERR_QUEUE	Error putting the thread on the ready queue

Store the thread entry point and parameter in the TCB. This may not be necessary for all architecture ports if they put all of this information in the initial thread stack.

Additional processing only required if stack-checking is enabled. Incurs a slight overhead on each thread creation and uses some additional storage in the TCB, but can be compiled out if not desired.

Call the arch-specific routine to set up the stack. This routine is responsible for creating the context save area necessary for allowing atomThreadSwitch() to schedule it in. The initial archContextSwitch() call when this thread gets scheduled in the first time will then restore the program counter to the thread entry point, and any other necessary register values ready for it to start running.

If the OS is started and we're in thread context, check if we should be scheduled in now.

References archThreadContextInit(), ATOM_ERR_PARAM, ATOM_ERR_QUEUE, ATOM_OK, atomCurrentContext(), atomOSStarted, atomSched(), CRITICAL_END, CRITICAL_START, CRITICAL_STORE, atom_tcb::entry_param, atom_tcb::entry_point, FALSE, atom_tcb::next_tcb, atom_tcb::prev_tcb, atom_tcb::priority, STACK_CHECK_BYTE, atom_tcb::suspend_timo_cb, atom_tcb::suspended, tcbEnqueuePriority(), TRUE, and uint8_t.

Referenced by atomOSInit().

tcbDequeueEntry

```
ATOM_TCB* tcbDequeueEntry  (  ATOM_TCB **   tcb_queue_ptr,
                              ATOM_TCB *    tcb_ptr
                           )
```

This is an internal function not for use by application code.

Dequeues a particular TCB from the queue pointed to by tcb_queue_ptr.

The TCB will be removed from the queue.

tcb_queue_ptr may be modified by the routine if the dequeued TCB was the list head. It is valid for tcb_queue_ptr to point to a NULL pointer, which is the case if the queue is currently empty. In this case the function returns NULL.

NOTE: Assumes that the caller is already in a critical section.

Parameters:

`[in,out]`	*tcb_queue_ptr*	Pointer to TCB queue head pointer
`[in]`	*tcb_ptr*	Pointer to TCB to dequeue

Returns:

Pointer to the dequeued TCB, or NULL if entry wasn't found

References atom_tcb::next_tcb, and atom_tcb::prev_tcb.

Referenced by atomMutexGet(), atomQueueGet(), atomQueuePut(), and atomSemGet().

tcbDequeueHead

ATOM_TCB* tcbDequeueHead (ATOM_TCB ** *tcb_queue_ptr*)

This is an internal function not for use by application code.

Dequeues the highest priority TCB on the queue pointed to by tcb_queue_ptr.

The TCB will be removed from the queue. Same priority TCBs are dequeued in FIFO order.

tcb_queue_ptr will be modified by the routine if a TCB is dequeued, as this will be the list head. It is valid for tcb_queue_ptr to point to a NULL pointer, which is the case if the queue is currently empty. In this case the function returns NULL.

NOTE: Assumes that the caller is already in a critical section.

Parameters:

`[in,out]`	*tcb_queue_ptr*	Pointer to TCB queue head pointer

Returns:

Pointer to highest priority TCB on queue, or NULL if queue empty

References atom_tcb::next_tcb, and atom_tcb::prev_tcb.

Referenced by atomMutexDelete(), atomMutexPut(), atomQueueDelete(), atomSched(), atomSemDelete(), and atomSemPut().

tcbDequeuePriority

```
ATOM_TCB* tcbDequeuePriority ( ATOM_TCB **  tcb_queue_ptr,
                               uint8_t      priority
                             )
```

This is an internal function not for use by application code.

Dequeues the first TCB of the given priority or higher, from the queue pointed to by tcb_queue_ptr. Because the queue is ordered high priority first, we only ever dequeue the list head, if any. If the list head is lower priority than we wish to dequeue, then all following ones will also be lower priority and hence are not parsed.

The TCB will be removed from the queue. Same priority TCBs will be dequeued in FIFO order.

tcb_queue_ptr may be modified by the routine if the dequeued TCB was the list head. It is valid for tcb_queue_ptr to point to a NULL pointer, which is the case if the queue is currently empty. In this case the function returns NULL.

NOTE: Assumes that the caller is already in a critical section.

Parameters:

`[in,out]`	*tcb_queue_ptr*	Pointer to TCB queue head pointer
`[in]`	*priority*	Minimum priority to qualify for dequeue

Returns:

Pointer to the dequeued TCB, or NULL if none found within priority

References atom_tcb::next_tcb, and atom_tcb::prev_tcb.

Referenced by atomOSStart(), and atomSched().

tcbEnqueuePriority

```
uint8_t tcbEnqueuePriority ( ATOM_TCB **  tcb_queue_ptr,
                             ATOM_TCB *   tcb_ptr
                           )
```

This is an internal function not for use by application code.

Enqueues the TCB tcb_ptr on the TCB queue pointed to by tcb_queue_ptr. TCBs are placed on the queue in priority order. If there are existing TCBs at the same priority as the TCB to be enqueued, the enqueued TCB will be placed at the end of the same-priority TCBs. Calls to tcbDequeuePriority() will dequeue same-priority TCBs in FIFO order.

tcb_queue_ptrmay be modified by the routine if the enqueued TCB becomes the new list head. It is valid for tcb_queue_ptr to point to a NULL pointer, which is the case if the queue is currently empty.

NOTE: Assumes that the caller is already in a critical section.

Parameters:

`[in,out]`	*tcb_queue_ptr*	Pointer to TCB queue head pointer
`[in]`	*tcb_ptr*	Pointer to TCB to enqueue

Return values:

ATOM_OK	Success
ATOM_ERR_PARAM	Bad parameters

References ATOM_ERR_PARAM, ATOM_OK, atom_tcb::next_tcb, atom_tcb::prev_tcb, atom_tcb::priority, and uint8_t.

Referenced by atomMutexDelete(), atomMutexGet(), atomMutexPut(), atomQueueDelete(), atomQueueGet(), atomQueuePut(), atomSched(), atomSemDelete(), atomSemGet(), atomSemPut(), and atomThreadCreate().

Variable Documentation

uint8_t atomOSStarted = FALSE

Set to TRUE when OS is started and running threads

Referenced by atomOSInit(), atomOSStart(), atomSched(), atomThreadCreate(), and atomTimerTick().

ATOM_TCB* tcbReadyQ = NULL

This is the head of the queue of threads that are ready to run. It is ordered by priority, with the higher priority threads coming first. Where there are multiple threads of the same priority, the TCB (task control block) pointers are FIFO-ordered.

Dequeuing the head is a fast operation because the list is ordered. Enqueuing may have to walk up to the end of the list. This means that context-switch times depend on the number of threads on the ready queue, but efficient use is made of available RAM

on tiny systems by avoiding priority tables etc. This scheme can be easily swapped out for other scheduler schemes by replacing the TCB enqueue and dequeue functions.

Once a thread is scheduled in, it is not present on the ready queue or any other kernel queue while it is running. When scheduled out it will be either placed back on the ready queue (if still ready), or will be suspended on some OS primitive if no longer ready (e.g. on the suspended TCB queue for a semaphore, or in the timer list if suspended on a timer delay).

Referenced by atomMutexDelete(), atomMutexPut(), atomQueueDelete(), atomSemDelete(), and atomSemPut().

atomsem.c File Reference

```
#include <stdio.h>
#include "atom.h"
#include "atomsem.h"
#include "atomtimer.h"
```

Data Structures

struct sem_timer

Typedefs

typedef struct sem_timer SEM_TIMER

Functions

uint8_t atomSemCreate (ATOM_SEM *sem, uint8_t initial_count)
uint8_t atomSemDelete (ATOM_SEM *sem)
uint8_t atomSemGet (ATOM_SEM *sem, int32_t timeout)
uint8_t atomSemPut (ATOM_SEM *sem)
uint8_t atomSemResetCount (ATOM_SEM *sem, uint8_t count)

Detailed Description

Semaphore library.

This module implements a counting semaphore library with the following features:

Flexible blocking APIs

Threads which wish to decrement a semaphore can choose whether to block, block with timeout, or not block if the semaphore has reached zero.

Interrupt-safe calls

All APIs can be called from interrupt context. Any calls which could potentially block have optional parameters to prevent blocking if you wish to call them from interrupt context. Any attempt to make a call which would block from interrupt context will be automatically and safely prevented.

Priority-based queueing

Where multiple threads are blocking on a semaphore, they are woken in order of the threads' priorities. Where multiple threads of the same priority are blocking, they are woken in FIFO order.

Count up to 255

Semaphore counts can be initialised and incremented up to a maximum of 255.

Smart semaphore deletion

Where a semaphore is deleted while threads are blocking on it, all blocking threads are woken and returned a status code to indicate the reason for being woken.

Usage instructions:

All semaphore objects must be initialised before use by calling atomSemCreate(). Once initialised atomSemGet() and atomSemPut() are used to decrement and increment the semaphore count respectively.

If a semaphore count reaches zero, further calls to atomSemGet() will block the calling thread (unless the calling parameters request no blocking). If a call is made to atomSemPut() while threads are blocking on a zero-count semaphore, the highest priority thread is woken. Where multiple threads of the same priority are blocking, they are woken in the order in which the threads started blocking.

A semaphore which is no longer required can be deleted using atomSemDelete(). This function automatically wakes up any threads which are waiting on the deleted semaphore.

Notes:

Note that those considering using a semaphore initialised to 1 for mutual exclusion purposes may wish to investigate the mutex library available in Atomthreads.

Typedef Documentation

typedef struct sem_timer SEM_TIMER

Function Documentation

atomSemCreate

```
uint8_t atomSemCreate ( ATOM_SEM *  sem,
                        uint8_t     initial_count
                      )
```

Initialises a semaphore object.

Must be called before calling any other semaphore library routines on a semaphore. Objects can be deleted later using atomSemDelete().

Does not allocate storage, the caller provides the semaphore object.

This function can be called from interrupt context.

Parameters:

[in]	*sem*	Pointer to semaphore object
[in]	*initial_count*	Initial count value

Return values:

ATOM_OK	Success
ATOM_ERR_PARAM	Bad parameters

References ATOM_ERR_PARAM, ATOM_OK, atom_sem::count, atom_sem::suspQ, and uint8_t.

atomSemDelete

uint8_t atomSemDelete (ATOM_SEM * *sem*)

Deletes a semaphore object.

Any threads currently suspended on the semaphore will be woken up with return status ATOM_ERR_DELETED. If called at thread context then the scheduler will be called during this function which may schedule in one of the woken threads depending on relative priorities.

This function can be called from interrupt context, but loops internally waking up all threads blocking on the semaphore, so the potential execution cycles cannot be determined in advance.

Parameters:

[in]	*sem*	Pointer to semaphore object

Return values:

ATOM_OK	Success
ATOM_ERR_QUEUE	Problem putting a woken thread on the ready queue
ATOM_ERR_TIMER	Problem cancelling a timeout on a woken thread

Only call the scheduler if we are in thread context, otherwise it will be called on exiting the ISR by atomIntExit().

References ATOM_ERR_DELETED, ATOM_ERR_PARAM, ATOM_ERR_QUEUE, ATOM_ERR_TIMER, ATOM_OK, atomCurrentContext(), atomSched(), atomTimerCancel(), CRITICAL_END, CRITICAL_START, CRITICAL_STORE, FALSE, atom_tcb::suspend_timo_cb,

atom_tcb::suspend_wake_status, atom_sem::suspQ, tcbDequeueHead(), tcbEnqueuePriority(), tcbReadyQ, TRUE, and uint8_t.

atomSemGet

```
uint8_t atomSemGet (  ATOM_SEM *   sem,
                      int32_t      timeout
                    )
```

Perform a get operation on a semaphore.

This decrements the current count value for the semaphore and returns. If the count value is already zero then the call will block until the count is incremented by another thread, or until the specified timeout is reached. Blocking threads will also be woken if the semaphore is deleted by another thread while blocking.

Depending on the timeout value specified the call will do one of the following if the count value is zero:

timeout == 0 : Call will block until the count is non-zero
timeout > 0 : Call will block until non-zero up to the specified timeout
timeout == -1 : Return immediately if the count is zero

If the call needs to block and timeout is zero, it will block indefinitely until atomSemPut() or atomSemDelete() is called on the semaphore.

If the call needs to block and timeout is non-zero, the call will only block for the specified number of system ticks after which time, if the thread was not already woken, the call will return with ATOM_TIMEOUT.

If the call would normally block and timeout is -1, the call will return immediately with ATOM_WOULDBLOCK.

This function can only be called from interrupt context if the timeout parameter is -1 (in which case it does not block).

Parameters:

`[in]`	*sem*	Pointer to semaphore object
`[in]`	*timeout*	Max system ticks to block (0 = forever)

Return values:

ATOM_OK	Success
ATOM_TIMEOUT	Semaphore timed out before being woken
ATOM_WOULDBLOCK	Called with timeout == -1 but count is zero
ATOM_ERR_DELETED	Semaphore was deleted while suspended
ATOM_ERR_CONTEXT	Not called in thread context and attempted to block

ATOM_ERR_PARAM	Bad parameter
ATOM_ERR_QUEUE	Problem putting the thread on the suspend queue
ATOM_ERR_TIMER	Problem registering the timeout

Store the timer details in the TCB so that we can cancel the timer callback if the semaphore is put before the timeout occurs.

Current thread now blocking, schedule in a new one. We already know we are in thread context so can call the scheduler from here.

Normal atomSemPut() wakeups will set ATOM_OK status, while timeouts will set ATOM_TIMEOUT and semaphore deletions will set ATOM_ERR_DELETED.

If we have been woken up with ATOM_OK then another thread incremented the semaphore and handed control to this thread. In theory the the posting thread increments the counter and as soon as this thread wakes up we decrement the counter here, but to prevent another thread preempting this thread and decrementing the semaphore before this section was scheduled back in, we emulate the increment and decrement by not incrementing in the atomSemPut() and not decrementing here. The count remains zero throughout preventing other threads preempting before we decrement the count again.

References ATOM_ERR_CONTEXT, ATOM_ERR_PARAM, ATOM_ERR_QUEUE, ATOM_ERR_TIMER, ATOM_OK, ATOM_WOULDBLOCK, atomCurrentContext(), atomSched(), atomTimerRegister(), atom_timer::cb_data, atom_timer::cb_func, atom_timer::cb_ticks, atom_sem::count, CRITICAL_END, CRITICAL_START, CRITICAL_STORE, FALSE, POINTER, sem_timer::sem_ptr, atom_tcb::suspend_timo_cb, atom_tcb::suspend_wake_status, atom_tcb::suspended, atom_sem::suspQ, sem_timer::tcb_ptr, tcbDequeueEntry(), tcbEnqueuePriority(), TRUE, and uint8_t.

atomSemPut

uint8_t atomSemPut (ATOM_SEM * *sem*)

Perform a put operation on a semaphore.

This increments the current count value for the semaphore and returns.

If the count value was previously zero and there are threads blocking on the semaphore, the call will wake up the highest priority thread suspended. Only one thread is woken per call to atomSemPut(). If multiple threads of the same priority are suspended, they are woken in order of suspension (FIFO).

This function can be called from interrupt context.

Parameters:

`[in]` *sem* Pointer to semaphore object

Return values:

ATOM_OK	Success
ATOM_ERR_OVF	The semaphore count would have overflowed (>255)
ATOM_ERR_PARAM	Bad parameter
ATOM_ERR_QUEUE	Problem putting a woken thread on the ready queue
ATOM_ERR_TIMER	Problem cancelling a timeout for a woken thread

Threads are woken up in priority order, with a FIFO system used on same priority threads. We always take the head, ordering is taken care of by an ordered list enqueue.

The scheduler may now make a policy decision to thread switch if we are currently in thread context. If we are in interrupt context it will be handled by atomIntExit().

References ATOM_ERR_OVF, ATOM_ERR_PARAM, ATOM_ERR_QUEUE, ATOM_ERR_TIMER, ATOM_OK, atomCurrentContext(), atomSched(), atomTimerCancel(), atom_sem::count, CRITICAL_END, CRITICAL_START, CRITICAL_STORE, FALSE, atom_tcb::suspend_timo_cb, atom_tcb::suspend_wake_status, atom_sem::suspQ, tcbDequeueHead(), tcbEnqueuePriority(), tcbReadyQ, and uint8_t.

atomSemResetCount

```
uint8_t atomSemResetCount ( ATOM_SEM *  sem,
                            uint8_t     count
                          )
```

Set a new count value on a semaphore.

Care must be taken when using this function, as there may be threads suspended on the semaphore. In general it should only be used once a semaphore is out of use.

This function can be called from interrupt context.

Parameters:

`[in]` *sem* Pointer to semaphore object
`[in]` *count* New count value

Return values:

ATOM_OK	Success
ATOM_ERR_PARAM	Bad parameter

References ATOM_ERR_PARAM, ATOM_OK, atom_sem::count, and uint8_t.

atommutex.c File Reference

```
#include <stdio.h>
#include "atom.h"
#include "atommutex.h"
#include "atomtimer.h"
```

Data Structures

struct mutex_timer

Typedefs

typedef struct mutex_timer MUTEX_TIMER

Functions

uint8_t atomMutexCreate (ATOM_MUTEX *mutex)
uint8_t atomMutexDelete (ATOM_MUTEX *mutex)
uint8_t atomMutexGet (ATOM_MUTEX *mutex, int32_t timeout)
uint8_t atomMutexPut (ATOM_MUTEX *mutex)

Detailed Description

Mutex library.

This module implements a mutual exclusion library with the following features:

Flexible blocking APIs

Threads which wish to take a mutex lock can choose whether to block, block with timeout, or not block if the mutex is already locked by a different thread.

Interrupt-safe calls

Some APIs can be called from interrupt context, but because a mutex must be owned by a particular thread the lock/unlock calls must be performed by threads only (i.e. it does not make sense to allow calls from interrupt context). All APIs are documented with their capability of being called from interrupt context. Any attempt to make a call which cannot be made from interrupt context will be automatically and safely prevented.

Priority-based queueing

Where multiple threads are blocking on a mutex, they are woken in order of the threads' priorities. Where multiple threads of the same priority are blocking, they are woken in FIFO order.

Recursive locks

A mutex can be locked recursively by the same thread up to a maximum recursion level of 255. An internal count of locks is maintained and the mutex is only released when the count reaches zero (when the thread has been unlocked the same number of times as it was locked). This makes a mutex more suitable for use as mutual exclusions than a semaphore with initial count of 1.

Thread ownership

Once a thread has locked a mutex, only that thread may release the lock. This is another feature which makes the mutex more suitable for mutual exclusion than a semaphore with initial count 1. It prevents programming errors whereby the wrong thread is used to perform the unlock. This cannot be done for semaphores which do not have a concept of ownership (because it must be possible to use them to signal between threads).

Smart mutex deletion

Where a mutex is deleted while threads are blocking on it, all blocking threads are woken and returned a status code to indicate the reason for being woken.

Usage instructions:

All mutex objects must be initialised before use by calling atomMutexCreate(). Once initialised atomMutexGet() and atomMutexPut() are used to lock and unlock the mutex respectively. A mutex may be locked recursively by the same thread, allowing for simplified code structure.

While a thread owns the lock on a mutex, no other thread can take the lock. These other threads will block until the mutex is released by the current owner (unless the calling parameters request no blocking, in which case the lock request will return with an error). If a mutex is released while threads are blocking on it, the highest priority thread is woken. Where multiple threads of the same priority are blocking, they are woken in the order in which the threads started blocking.

A mutex which is no longer required can be deleted using atomMutexDelete(). This function automatically wakes up any threads which are waiting on the deleted mutex.

Typedef Documentation

typedef struct mutex_timer MUTEX_TIMER

Function Documentation

atomMutexCreate

uint8_t atomMutexCreate (ATOM_MUTEX * *mutex*)

Initialises a mutex object.

Must be called before calling any other mutex library routines on a mutex. Objects can be deleted later using atomMutexDelete().

Does not set the owner of a mutex. atomMutexGet() must be called after creation in order to actually take ownership.

Does not allocate storage, the caller provides the mutex object.

This function can be called from interrupt context.

Parameters:

`[in]` *mutex* Pointer to mutex object

Return values:

ATOM_OK	Success
ATOM_ERR_PARAM	Bad parameters

References ATOM_ERR_PARAM, ATOM_OK, atom_mutex::count, atom_mutex::owner, atom_mutex::suspQ, and uint8_t.

atomMutexDelete

uint8_t atomMutexDelete (ATOM_MUTEX * *mutex*)

Deletes a mutex object.

Any threads currently suspended on the mutex will be woken up with return status ATOM_ERR_DELETED. If called at thread context then the scheduler will be called during this function which may schedule in one of the woken threads depending on relative priorities.

This function can be called from interrupt context, but loops internally waking up all threads blocking on the mutex, so the potential execution cycles cannot be determined in advance.

Parameters:

`[in]` *mutex* Pointer to mutex object

Return values:

ATOM_OK	Success

ATOM_ERR_QUEUE	Problem putting a woken thread on the ready queue
ATOM_ERR_TIMER	Problem cancelling a timeout on a woken thread

Only call the scheduler if we are in thread context, otherwise it will be called on exiting the ISR by atomIntExit().

References ATOM_ERR_DELETED, ATOM_ERR_PARAM, ATOM_ERR_QUEUE, ATOM_ERR_TIMER, ATOM_OK, atomCurrentContext(), atomSched(), atomTimerCancel(), CRITICAL_END, CRITICAL_START, CRITICAL_STORE, FALSE, atom_tcb::suspend_timo_cb, atom_tcb::suspend_wake_status, atom_mutex::suspQ, tcbDequeueHead(), tcbEnqueuePriority(), tcbReadyQ, TRUE, and uint8_t.

atomMutexGet

```
uint8_t atomMutexGet ( ATOM_MUTEX *  mutex,
                       int32_t       timeout
                     )
```

Take the lock on a mutex.

This takes ownership of a mutex if it is not currently owned. Ownership is held by this thread until a corresponding call to atomMutexPut() by the same thread.

Can be called recursively by the original locking thread (owner). Recursive calls are counted, and ownership is not relinquished until the number of unlock (atomMutexPut()) calls by the owner matches the number of lock (atomMutexGet()) calls.

No thread other than the owner can lock or unlock the mutex while it is locked by another thread.

Depending on the timeout value specified the call will do one of the following if the mutex is already locked by another thread:

timeout == 0 : Call will block until the mutex is available
timeout > 0 : Call will block until available up to the specified timeout
timeout == -1 : Return immediately if mutex is locked by another thread

If the call needs to block and timeout is zero, it will block indefinitely until the owning thread calls atomMutexPut() or atomMutexDelete() is called on the mutex.

If the call needs to block and timeout is non-zero, the call will only block for the specified number of system ticks after which time, if the thread was not already woken, the call will return with ATOM_TIMEOUT.

If the call would normally block and timeout is -1, the call will return immediately with ATOM_WOULDBLOCK.

This function can only be called from thread context. A mutex has the concept of an owner thread, so it is never valid to make a mutex call from interrupt context when there is no thread to associate with.

Parameters:

`[in]`	*mutex*	Pointer to mutex object
`[in]`	*timeout*	Max system ticks to block (0 = forever)

Return values:

ATOM_OK	Success
ATOM_TIMEOUT	Mutex timed out before being woken
ATOM_WOULDBLOCK	Called with timeout == -1 but count is zero
ATOM_ERR_DELETED	Mutex was deleted while suspended
ATOM_ERR_CONTEXT	Not called in thread context and attempted to block
ATOM_ERR_PARAM	Bad parameter
ATOM_ERR_QUEUE	Problem putting the thread on the suspend queue
ATOM_ERR_TIMER	Problem registering the timeout
ATOM_ERR_OVF	The recursive lock count would have overflowed (>255)

Check we are at thread context. Because mutexes have the concept of owner threads, it is never valid to call here from an ISR, regardless of whether we will block.

Store the timer details in the TCB so that we can cancel the timer callback if the mutex is put before the timeout occurs.

Current thread now blocking, schedule in a new one. We already know we are in thread context so can call the scheduler from here.

Normal atomMutexPut() wakeups will set ATOM_OK status, while timeouts will set ATOM_TIMEOUT and mutex deletions will set ATOM_ERR_DELETED.

If we were woken up by another thread relinquishing the mutex and handing this thread ownership, then the relinquishing thread will set status to ATOM_OK and will make this thread the owner. Setting the owner before waking the thread ensures that no other thread can preempt and take ownership of the mutex between this thread being made ready to run, and actually being scheduled back in here.

Since this thread has just gained ownership, the lock count is zero and should be incremented once for this call.

References ATOM_ERR_CONTEXT, ATOM_ERR_OVF, ATOM_ERR_PARAM, ATOM_ERR_QUEUE, ATOM_ERR_TIMER, ATOM_OK, ATOM_WOULDBLOCK, atomCurrentContext(), atomSched(), atomTimerRegister(), atom_timer::cb_data, atom_timer::cb_func, atom_timer::cb_ticks, atom_mutex::count, CRITICAL_END, CRITICAL_START, CRITICAL_STORE, FALSE, mutex_timer::mutex_ptr, atom_mutex::owner, POINTER, atom_tcb::suspend_timo_cb, atom_tcb::suspend_wake_status, atom_tcb::suspended, atom_mutex::suspQ, mutex_timer::tcb_ptr, tcbDequeueEntry(), tcbEnqueuePriority(), TRUE, and uint8_t.

atomMutexPut

uint8_t atomMutexPut (ATOM_MUTEX * *mutex*)

Give back the lock on a mutex.

This checks that the mutex is owned by the calling thread, and decrements the recursive lock count. Once the lock count reaches zero, the lock is considered relinquished and no longer owned by this thread.

If the lock is relinquished and there are threads blocking on the mutex, the call will wake up the highest priority thread suspended. Only one thread is woken per call to atomMutexPut(). If multiple threads of the same priority are suspended, they are woken in order of suspension (FIFO).

This function can only be called from thread context. A mutex has the concept of an owner thread, so it is never valid to make a mutex call from interrupt context when there is no thread to associate with.

Parameters:

`[in]` *mutex* Pointer to mutex object

Return values:

ATOM_OK	Success
ATOM_ERR_PARAM	Bad parameter
ATOM_ERR_QUEUE	Problem putting a woken thread on the ready queue
ATOM_ERR_TIMER	Problem cancelling a timeout for a woken thread
ATOM_ERR_OWNERSHIP	Attempt to unlock mutex not owned by this thread

Threads are woken up in priority order, with a FIFO system used on same priority threads. We always take the head, ordering is taken care of by an ordered list enqueue.

The scheduler may now make a policy decision to thread switch. We already know we are in thread context so can call the scheduler from here.

Relinquished ownership and no threads waiting. Nothing to do.

Decremented lock but still retain ownership due to recursion. Nothing to do.

References ATOM_ERR_OWNERSHIP, ATOM_ERR_PARAM, ATOM_ERR_QUEUE, ATOM_ERR_TIMER, ATOM_OK, atomCurrentContext(), atomSched(), atomTimerCancel(), atom_mutex::count, CRITICAL_END, CRITICAL_START, CRITICAL_STORE, FALSE, atom_mutex::owner, atom_tcb::suspend_timo_cb, atom_tcb::suspend_wake_status, atom_mutex::suspQ, tcbDequeueHead(), tcbEnqueuePriority(), tcbReadyQ, and uint8_t.

atomqueue.c File Reference

```
#include <stdio.h>
#include <string.h>
#include "atom.h"
#include "atomqueue.h"
#include "atomtimer.h"
```

Data Structures

struct queue_timer

Typedefs

typedef
struct queue_timer QUEUE_TIMER

Functions

uint8_t atomQueueCreate (ATOM_QUEUE *qptr, uint8_t *buff_ptr, uint32_t unit_size, uint32_t max_num_msgs)
uint8_t atomQueueDelete (ATOM_QUEUE *qptr)
uint8_t atomQueueGet (ATOM_QUEUE *qptr, int32_t timeout, uint8_t *msgptr)
uint8_t atomQueuePut (ATOM_QUEUE *qptr, int32_t timeout, uint8_t *msgptr)

Detailed Description

Queue library.

This module implements a queue / message-passing library with the following features:

Flexible blocking APIs

Threads which wish to make a call which may block can choose whether to block, block with timeout, or not block and return a relevent status code.

Interrupt-safe calls

All APIs can be called from interrupt context. Any calls which could potentially block have optional parameters to prevent blocking if you wish to call them from interrupt context. Any attempt to make a call which would block from interrupt context will be automatically and safely prevented.

Priority-based queueing

Where multiple threads are blocking on a queue, they are woken in order of the threads' priorities. Where multiple threads of the same priority are blocking, they are woken in FIFO order.

Configurable queue sizes

Queues can be created with any sized message, and any number of stored messages.

Smart queue deletion

Where a queue is deleted while threads are blocking on it, all blocking threads are woken and returned a status code to indicate the reason for being woken.

Usage instructions:

All queue objects must be initialised before use by calling atomQueueCreate(). Once initialised atomQueueGet() and atomQueuePut() are used to send and receive messages via the queue respectively.

Messages can be added to a queue by calling atomQueuePut(). If the queue is full the caller will block until space becomes available (by a message being removed from the queue). Optionally a non-blocking call can be made in which case the call will return with a status code indicating that the queue is full. This allows messages to be posted from interrupt handlers or threads which you do not wish to block, providing it is not fatal that the call could fail if the queue was full.

Messages can be received from the queue by calling atomQueueGet(). This will return the first message available in the queue in FIFO order. If the queue is empty then the call will block. Optionally, a non-blocking call can be made in which case the call will return with a status code indicating that the queue is full. This allows messages to be received by interrupt handlers or threads which you do not wish to block.

A queue which is no longer required can be deleted using atomQueueDelete(). This function automatically wakes up any threads which are waiting on the deleted queue.

Typedef Documentation

typedef struct queue_timer QUEUE_TIMER

Function Documentation

atomQueueCreate

```
uint8_t atomQueueCreate ( ATOM_QUEUE *  qptr,
                          uint8_t *     buff_ptr,
                          uint32_t      unit_size,
                          uint32_t      max_num_msgs
                        )
```

Initialises a queue object.

Must be called before calling any other queue library routines on a queue. Objects can be deleted later using atomQueueDelete().

Does not allocate storage, the caller provides the queue object.

Callers pass in their own buffer area for storing the queue messages while in transit between threads. The provided storage must be large enough to store (unit_size * max_num_mgs) bytes. i.e. the storage area will be used for up to max_num_msgs messages each of size unit_size.

Queues use a fixed-size message.

This function can be called from interrupt context.

Parameters:

[in]	*qptr*	Pointer to queue object
[in]	*buff_ptr*	Pointer to buffer storage area
[in]	*unit_size*	Size in bytes of each queue message
[in]	*max_num_msgs*	Maximum number of messages in the queue

Return values:

ATOM_OK	Success
ATOM_ERR_PARAM	Bad parameters

References ATOM_ERR_PARAM, ATOM_OK, atom_queue::buff_ptr, atom_queue::getSuspQ, atom_queue::insert_index, atom_queue::max_num_msgs, atom_queue::num_msgs_stored, atom_queue::putSuspQ, atom_queue::remove_index, uint8_t, and atom_queue::unit_size.

atomQueueDelete

```
uint8_t atomQueueDelete ( ATOM_QUEUE *  qptr  )
```

Deletes a queue object.

Any threads currently suspended on the queue will be woken up with return status ATOM_ERR_DELETED. If called at thread context then the scheduler will be called during this function which may schedule in one of the woken threads depending on relative priorities.

This function can be called from interrupt context, but loops internally waking up all threads blocking on the queue, so the potential execution cycles cannot be determined in advance.

Parameters:

`[in]` *qptr* Pointer to queue object

Return values:

ATOM_OK	Success
ATOM_ERR_QUEUE	Problem putting a woken thread on the ready queue
ATOM_ERR_TIMER	Problem cancelling a timeout on a woken thread

Only call the scheduler if we are in thread context, otherwise it will be called on exiting the ISR by atomIntExit().

References ATOM_ERR_DELETED, ATOM_ERR_PARAM, ATOM_ERR_QUEUE, ATOM_ERR_TIMER, ATOM_OK, atomCurrentContext(), atomSched(), atomTimerCancel(), CRITICAL_END, CRITICAL_START, CRITICAL_STORE, FALSE, atom_queue::getSuspQ, atom_queue::putSuspQ, atom_tcb::suspend_timo_cb, atom_tcb::suspend_wake_status, tcbDequeueHead(), tcbEnqueuePriority(), tcbReadyQ, TRUE, and uint8_t.

atomQueueGet

```
uint8_t atomQueueGet (  ATOM_QUEUE *   qptr,
                        int32_t        timeout,
                        uint8_t *      msgptr
                      )
```

Attempt to retrieve a message from a queue.

Retrieves one message at a time. Messages are copied into the passed msgptr storage area which should be large enough to contain one message of unit_size bytes. Where multiple messages are in the queue, messages are retrieved in FIFO order.

If the queue is currently empty, the call will do one of the following depending on the timeout value specified:

timeout == 0 : Call will block until a message is available
timeout > 0 : Call will block until a message or the specified timeout
timeout == -1 : Return immediately if no message is on the queue

If a maximum timeout value is specified (timeout > 0), and no message is present on the queue for the specified number of system ticks, the call will return with ATOM_TIMEOUT.

This function can only be called from interrupt context if the timeout parameter is -1 (in which case it does not block).

Parameters:

`[in]`	*qptr*	Pointer to queue object
`[in]`	*timeout*	Max system ticks to block (0 = forever, -1 = no block)
`[out]`	*msgptr*	Pointer to which the received message will be copied

Return values:

ATOM_OK	Success
ATOM_TIMEOUT	Queue wait timed out before being woken
ATOM_WOULDBLOCK	Called with timeout == -1 but queue was empty
ATOM_ERR_DELETED	Queue was deleted while suspended
ATOM_ERR_CONTEXT	Not called in thread context and attempted to block
ATOM_ERR_PARAM	Bad parameter
ATOM_ERR_QUEUE	Problem putting the thread on the suspend queue
ATOM_ERR_TIMER	Problem registering the timeout

Fill out the data needed by the callback to wake us up.

Store the timer details in the TCB so that we can cancel the timer callback if the queue is put before the timeout occurs.

Current thread now blocking, schedule in a new one. We already know we are in thread context so can call the scheduler from here.

Normal atomQueuePut() wakeups will set ATOM_OK status, while timeouts will set ATOM_TIMEOUT and queue deletions will set ATOM_ERR_DELETED.

Check suspend_wake_status. If it is ATOM_OK then we were woken because a message has been put on the queue and we can now copy it out. Otherwise we were woken because we timed out waiting for a message, or the queue was deleted, so we should just quit.

The scheduler may now make a policy decision to thread switch if we are currently in thread context. If we are in interrupt context it will be handled by atomIntExit().

References ATOM_ERR_CONTEXT, ATOM_ERR_PARAM, ATOM_ERR_QUEUE, ATOM_ERR_TIMER, ATOM_OK, ATOM_WOULDBLOCK, atomCurrentContext(), atomSched(), atomTimerRegister(), atom_timer::cb_data, atom_timer::cb_func, atom_timer::cb_ticks, CRITICAL_END, CRITICAL_START, CRITICAL_STORE, FALSE, atom_queue::getSuspQ, atom_queue::num_msgs_stored, POINTER, queue_timer::queue_ptr, atom_tcb::suspend_timo_cb, atom_tcb::suspend_wake_status, atom_tcb::suspended, queue_timer::suspQ, queue_timer::tcb_ptr, tcbDequeueEntry(), tcbEnqueuePriority(), TRUE, and uint8_t.

atomQueuePut

```
uint8_t atomQueuePut ( ATOM_QUEUE *  qptr,
                       int32_t       timeout,
                       uint8_t *     msgptr
                     )
```

Attempt to put a message onto a queue.

Sends one message at a time. Messages are copied from the passed msgptr storage area which should contain a message of unit_size bytes.

If the queue is currently full, the call will do one of the following depending on the timeout value specified:

timeout == 0 : Call will block until space is available
timeout > 0 : Call will block until space or the specified timeout
timeout == -1 : Return immediately if the queue is full

If a maximum timeout value is specified (timeout > 0), and no space is available on the queue for the specified number of system ticks, the call will return with ATOM_TIMEOUT.

This function can only be called from interrupt context if the timeout parameter is -1 (in which case it does not block and may fail to post a message if the queue is full).

Parameters:

`[in]`	*qptr*	Pointer to queue object
`[in]`	*timeout*	Max system ticks to block (0 = forever, -1 = no block)
`[out]`	*msgptr*	Pointer from which the message should be copied out

Return values:

ATOM_OK	Success
ATOM_WOULDBLOCK	Called with timeout == -1 but queue was full
ATOM_TIMEOUT	Queue wait timed out before being woken
ATOM_ERR_DELETED	Queue was deleted while suspended
ATOM_ERR_CONTEXT	Not called in thread context and attempted to block
ATOM_ERR_PARAM	Bad parameter
ATOM_ERR_QUEUE	Problem putting the thread on the suspend queue
ATOM_ERR_TIMER	Problem registering the timeout

Fill out the data needed by the callback to wake us up.

Store the timer details in the TCB so that we can cancel the timer callback if a message is removed from the queue before the timeout occurs.

Current thread now blocking, schedule in a new one. We already know we are in thread context so can call the scheduler from here.

Normal atomQueueGet() wakeups will set ATOM_OK status, while timeouts will set ATOM_TIMEOUT and queue deletions will set ATOM_ERR_DELETED.

Check suspend_wake_status. If it is ATOM_OK then we were woken because a message has been removed from the queue and we can now add ours. Otherwise we were woken because we timed out waiting for a message, or the queue was deleted, so we should just quit.

The scheduler may now make a policy decision to thread switch if we are currently in thread context. If we are in interrupt context it will be handled by atomIntExit().

References ATOM_ERR_CONTEXT, ATOM_ERR_PARAM, ATOM_ERR_QUEUE, ATOM_ERR_TIMER, ATOM_OK, ATOM_WOULDBLOCK, atomCurrentContext(), atomSched(), atomTimerRegister(), atom_timer::cb_data, atom_timer::cb_func, atom_timer::cb_ticks, CRITICAL_END, CRITICAL_START, CRITICAL_STORE, FALSE, atom_queue::max_num_msgs, atom_queue::num_msgs_stored, POINTER, atom_queue::putSuspQ, queue_timer::queue_ptr, atom_tcb::suspend_timo_cb, atom_tcb::suspend_wake_status, atom_tcb::suspended, queue_timer::suspQ, queue_timer::tcb_ptr, tcbDequeueEntry(), tcbEnqueuePriority(), TRUE, and uint8_t.

atomtimer.c File Reference

```
#include <stdio.h>
#include "atom.h"
```

Data Structures

struct delay_timer

Typedefs

typedef struct delay_timer DELAY_TIMER

Functions

uint8_t atomTimerRegister (ATOM_TIMER *timer_ptr)
uint8_t atomTimerCancel (ATOM_TIMER *timer_ptr)
uint32_t atomTimeGet (void)
void atomTimeSet (uint32_t new_time)
void atomTimerTick (void)
uint8_t atomTimerDelay (uint32_t ticks)

Detailed Description

Timer and system tick library.

This module implements kernel system tick / clock functionality and timer functionality for kernel and application code.

Timer callbacks

Application and kernel code uses this module to request callbacks at a specific number of system ticks in the future. atomTimerRegister() can be called with a structure filled out requesting callbacks in a specific number of ticks. When the timer expires the requested callback function is called.

Thread delays

Application threads can use atomTimerDelay() to request that the thread delay for the specified number of system ticks. The thread will be put in the timer list and taken off the ready queue. When the timer expires the thread will be made ready-to-run again. This internally uses the same atomTimerRegister() function that is used for registering all timers.

System tick / Clock

This module also implements the system tick. At a predefined interval (SYSTEM_TICKS_PER_SEC) architecture ports arrange
for atomTimerTick() to be called. The tick increments the system tick count, which can be queried by application code using atomTimeGet(). On this tick, the

registered timer list is checked for any timers which have expired. Those which have expired have their callback functions called. It is also on this system tick that round-robin rescheduling time-slices occur. On exit from the tick interrupt handler the kernel checks whether there are two or more threads ready-to-run at the same priority, and if so uses round-robin to schedule in the next thread. This is in contrast to other (non-timer-tick) interrupts which do not allow for round-robin rescheduling to occur, as they should only occur on a new timer tick.

Typedef Documentation

typedef struct delay_timer DELAY_TIMER

Function Documentation

atomTimeGet

uint32_t atomTimeGet (void)

Returns the current system tick time.

This function can be called from interrupt context.

Return values:

Current system tick count

atomTimerCancel

uint8_t atomTimerCancel (ATOM_TIMER * *timer_ptr*)

Cancel a timer callback previously registered using atomTimerRegister().

This function can be called from interrupt context, but loops internally through the time list, so the potential execution cycles cannot be determined in advance.

Parameters:

`[in]` *timer_ptr* Pointer to timer to cancel

Return values:

ATOM_OK	Success
ATOM_ERR_PARAM	Bad parameters
ATOM_ERR_NOT_FOUND	Timer registration was not found

References ATOM_ERR_NOT_FOUND, ATOM_ERR_PARAM, ATOM_OK, CRITICAL_END, CRITICAL_START, CRITICAL_STORE, atom_timer::next_timer, and uint8_t.

Referenced by atomMutexDelete(), atomMutexPut(), atomQueueDelete(), atomSemDelete(), and atomSemPut().

atomTimerDelay

uint8_t atomTimerDelay (uint32_t *ticks*)

Suspend a thread for the given number of system ticks.

Note that the wakeup time is the number of ticks from the current system tick, therefore, for a one tick delay, the thread may be woken up at any time between the atomTimerDelay() call and the next system tick. For a minimum number of ticks, you should specify minimum number of ticks + 1.

This function can only be called from thread context.

Parameters:

`[in]` *ticks* Number of system ticks to delay (must be > 0)

Return values:

ATOM_OK	Successful delay
ATOM_ERR_PARAM	Bad parameter (ticks must be non-zero)
ATOM_ERR_CONTEXT	Not called from thread context

References ATOM_ERR_CONTEXT, ATOM_ERR_PARAM, ATOM_ERR_TIMER, ATOM_OK, atomCurrentContext(), atomSched(), atomTimerRegister(), atom_timer::cb_data, atom_timer::cb_func, atom_timer::cb_ticks, CRITICAL_END, CRITICAL_START, CRITICAL_STORE, FALSE, POINTER, atom_tcb::suspend_timo_cb, atom_tcb::suspended, delay_timer::tcb_ptr, TRUE, and uint8_t.

atomTimerRegister

uint8_t atomTimerRegister (ATOM_TIMER * *timer_ptr*)

Register a timer callback.

Callers should fill out and pass in a timer descriptor, containing the number of system ticks until they would like a callback, together with a callback function and optional parameter. The number of ticks must be greater than zero.

On the relevant system tick count, the callback function will be called.

These timers are used by some of the OS library routines, but they can also be used by application code requiring timer facilities at system tick resolution.

This function can be called from interrupt context, but loops internally through the time list, so the potential execution cycles cannot be determined in advance.

Parameters:

`[in]` *timer_ptr* Pointer to timer descriptor

Return values:

ATOM_OK	Success
ATOM_ERR_PARAM	Bad parameters

References ATOM_ERR_PARAM, ATOM_OK, atom_timer::cb_func, atom_timer::cb_ticks, CRITICAL_END, CRITICAL_START, CRITICAL_STORE, atom_timer::next_timer, and uint8_t.

Referenced by atomMutexGet(), atomQueueGet(), atomQueuePut(), atomSemGet(), and atomTimerDelay().

atomTimerTick

void atomTimerTick (void)

System tick handler.

User ports are responsible for calling this routine once per system tick.

On each system tick this routine is called to do the following: 1. Increase the system tick count 2. Call back to any registered timer callbacks

Returns: None
References atomOSStarted.

atomTimeSet

void atomTimeSet (uint32_t *new_time*)

This is an internal function not for use by application code.

Sets the current system tick time.

Currently only required for automated test suite to test clock behaviour.

This function can be called from interrupt context.

Parameters:

`[in]` *new_time* New system tick time value

Returns:

None

Here is a list of all functions, variables, defines, enums, and typedefs with links to the files they belong to:

- a -

- archContextSwitch() : atom.h
- archFirstThreadRestore() : atom.h
- archThreadContextInit() : atom.h
- ATOM_ERR_CONTEXT : atom.h
- ATOM_ERR_DELETED : atom.h
- ATOM_ERR_NOT_FOUND : atom.h
- ATOM_ERR_OVF : atom.h
- ATOM_ERR_OWNERSHIP : atom.h
- ATOM_ERR_PARAM : atom.h
- ATOM_ERR_QUEUE : atom.h
- ATOM_ERR_TIMER : atom.h
- ATOM_ERROR : atom.h
- ATOM_MUTEX : atommutex.h
- ATOM_OK : atom.h
- ATOM_QUEUE : atomqueue.h
- ATOM_SEM : atomsem.h
- ATOM_TCB : atom.h
- ATOM_TIMEOUT : atom.h
- ATOM_TIMER : atomtimer.h
- ATOM_WOULDBLOCK : atom.h
- atomCurrentContext() : atom.h , atomkernel.c
- atomIntEnter() : atom.h , atomkernel.c
- atomIntExit() : atom.h , atomkernel.c
- atomMutexCreate() : atommutex.c , atommutex.h
- atomMutexDelete() : atommutex.c , atommutex.h
- atomMutexGet() : atommutex.h , atommutex.c
- atomMutexPut() : atommutex.c , atommutex.h
- atomOSInit() : atom.h , atomkernel.c
- atomOSStart() : atom.h , atomkernel.c
- atomOSStarted : atom.h , atomkernel.c
- atomQueueCreate() : atomqueue.c , atomqueue.h

- atomQueueDelete() : atomqueue.c , atomqueue.h
- atomQueueGet() : atomqueue.c , atomqueue.h
- atomQueuePut() : atomqueue.h , atomqueue.c
- atomSched() : atom.h , atomkernel.c
- atomSemCreate() : atomsem.c , atomsem.h
- atomSemDelete() : atomsem.h , atomsem.c
- atomSemGet() : atomsem.c , atomsem.h
- atomSemPut() : atomsem.c , atomsem.h
- atomSemResetCount() : atomsem.c , atomsem.h
- atomThreadCreate() : atomkernel.c , atom.h
- atomThreadStackCheck() : atom.h
- atomTimeGet() : atomtimer.c , atomtimer.h
- atomTimerCancel() : atomtimer.h , atomtimer.c
- atomTimerDelay() : atomtimer.c , atomtimer.h
- atomTimerRegister() : atomtimer.c , atomtimer.h
- atomTimerTick() : atomtimer.c , atom.h
- atomTimeSet() : atomtimer.c , atomtimer.h

- c -

- CRITICAL_END : atomport-template.h
- CRITICAL_START : atomport-template.h
- CRITICAL_STORE : atomport-template.h

- d -

- DELAY_TIMER : atomtimer.c

- f -

- FALSE : atom.h

- i -

- IDLE_THREAD_PRIORITY : atom.h
- int16_t : atomport-template.h
- int32_t : atomport-template.h
- int64_t : atomport-template.h
- int8_t : atomport-template.h

- m -
    - MUTEX_TIMER : atommutex.c

- p -
    - POINTER : atomport-template.h

- q -
    - QUEUE_TIMER : atomqueue.c

- s -
    - SEM_TIMER : atomsem.c
    - STACK_CHECK_BYTE : atomkernel.c
    - SYSTEM_TICKS_PER_SEC : atomport-template.h

- t -
    - tcbDequeueEntry() : atom.h , atomkernel.c
    - tcbDequeueHead() : atomkernel.c , atom.h
    - tcbDequeuePriority() : atom.h , atomkernel.c
    - tcbEnqueuePriority() : atomkernel.c , atom.h
    - tcbReadyQ : atomkernel.c , atom.h
    - TIMER_CB_FUNC : atomtimer.h
    - TRUE : atom.h

- u -
    - uint16_t : atomport-template.h
    - uint32_t : atomport-template.h
    - uint64_t : atomport-template.h
    - uint8_t : atomport-template.h

---

atom_mutex Struct Reference

`#include <atommutex.h>`

Data Fields

ATOM_TCB * suspQ
ATOM_TCB * owner
uint8_t count

Field Documentation

uint8_t atom_mutex::count
Referenced by atomMutexCreate(), atomMutexGet(), and atomMutexPut().

ATOM_TCB* atom_mutex::owner
Referenced by atomMutexCreate(), atomMutexGet(), and atomMutexPut().

ATOM_TCB* atom_mutex::suspQ
Referenced by atomMutexCreate(), atomMutexDelete(), atomMutexGet(), and atomMutexPut().

atom_queue Struct Reference

`#include <atomqueue.h>`

Data Fields

ATOM_TCB * putSuspQ
ATOM_TCB * getSuspQ
uint8_t * buff_ptr
uint32_t unit_size
uint32_t max_num_msgs
uint32_t insert_index
uint32_t remove_index
uint32_t num_msgs_stored

Field Documentation

uint8_t* atom_queue::buff_ptr
Referenced by atomQueueCreate().

ATOM_TCB* atom_queue::getSuspQ
Referenced by atomQueueCreate(), atomQueueDelete(), and atomQueueGet().

uint32_t atom_queue::insert_index
Referenced by atomQueueCreate().

uint32_t atom_queue::max_num_msgs
Referenced by atomQueueCreate(), and atomQueuePut().

uint32_t atom_queue::num_msgs_stored
Referenced by atomQueueCreate(), atomQueueGet(), and atomQueuePut().

ATOM_TCB* atom_queue::putSuspQ
Referenced by atomQueueCreate(), atomQueueDelete(), and atomQueuePut().

uint32_t atom_queue::remove_index
Referenced by atomQueueCreate().

uint32_t atom_queue::unit_size
Referenced by atomQueueCreate().

atom_sem Struct Reference

`#include <atomsem.h>`

Data Fields

ATOM_TCB * suspQ
uint8_t count

Field Documentation

uint8_t atom_sem::count
Referenced by atomSemCreate(), atomSemGet(), atomSemPut(), and atomSemResetCount().

ATOM_TCB* atom_sem::suspQ
Referenced by atomSemCreate(), atomSemDelete(), atomSemGet(), and atomSemPut()

atom_tcb Struct Reference

`#include <atom.h>`

Data Fields

POINTER sp_save_ptr
uint8_t priority
void(* entry_point)
(uint32_t)
uint32_t entry_param
struct atom_tcb * prev_tcb
struct atom_tcb * next_tcb
uint8_t suspended

uint8_t suspend_wake_status
ATOM_TIMER * suspend_timo_cb

Field Documentation

uint32_t atom_tcb::entry_param
Referenced by atomThreadCreate().

void(* atom_tcb::entry_point)(uint32_t)
Referenced by atomThreadCreate().

struct atom_tcb* atom_tcb::next_tcb [read]
Referenced by atomThreadCreate(), tcbDequeueEntry(), tcbDequeueHead(), tcbDequeuePriority(), and tcbEnqueuePriority().

struct atom_tcb* atom_tcb::prev_tcb [read]
Referenced by atomThreadCreate(), tcbDequeueEntry(), tcbDequeueHead(), tcbDequeuePriority(), and tcbEnqueuePriority().

uint8_t atom_tcb::priority
Referenced by atomSched(), atomThreadCreate(), and tcbEnqueuePriority().

POINTER atom_tcb::sp_save_ptr

ATOM_TIMER* atom_tcb::suspend_timo_cb
Referenced by atomMutexDelete(), atomMutexGet(), atomMutexPut(), atomQueueDelete(), atomQueueGet(), atomQueuePut(), atomSemDelete(), atomSemGet(), atomSemPut(), atomThreadCreate(), and atomTimerDelay().

uint8_t atom_tcb::suspend_wake_status
Referenced by atomMutexDelete(), atomMutexGet(), atomMutexPut(), atomQueueDelete(), atomQueueGet(), atomQueuePut(), atomSemDelete(), atomSemGet(), and atomSemPut().

uint8_t atom_tcb::suspended
Referenced by atomMutexGet(), atomQueueGet(), atomQueuePut(), atomSched(), atomSemGet(), atomThreadCreate(), and atomTimerDelay().

atom_timer Struct Reference

`#include <atomtimer.h>`

Data Fields

TIMER_CB_FUNC cb_func
POINTER cb_data
uint32_t cb_ticks
struct atom_timer * next_timer

Field Documentation

POINTER atom_timer::cb_data
Referenced by atomMutexGet(), atomQueueGet(), atomQueuePut(), atomSemGet(), and atomTimerDelay().

TIMER_CB_FUNC atom_timer::cb_func
Referenced by atomMutexGet(), atomQueueGet(), atomQueuePut(), atomSemGet(), atomTimerDelay(), and atomTimerRegister().

uint32_t atom_timer::cb_ticks
Referenced by atomMutexGet(), atomQueueGet(), atomQueuePut(), atomSemGet(), atomTimerDelay(), and atomTimerRegister().

struct atom_timer* atom_timer::next_timer `[read]`
Referenced by atomTimerCancel(), and atomTimerRegister().

delay_timer Struct Reference

Data Fields

ATOM_TCB * tcb_ptr

Field Documentation

ATOM_TCB* delay_timer::tcb_ptr
Referenced by atomTimerDelay().

mutex_timer Struct Reference

Data Fields

ATOM_TCB * tcb_ptr
ATOM_MUTEX * mutex_ptr

Field Documentation

ATOM_MUTEX* mutex_timer::mutex_ptr
Referenced by atomMutexGet().

ATOM_TCB* mutex_timer::tcb_ptr
Referenced by atomMutexGet().

queue_timer Struct Reference

Data Fields

ATOM_TCB * tcb_ptr
ATOM_QUEUE * queue_ptr
ATOM_TCB ** suspQ

Field Documentation

ATOM_QUEUE* queue_timer::queue_ptr
Referenced by atomQueueGet(), and atomQueuePut().

ATOM_TCB** queue_timer::suspQ
Referenced by atomQueueGet(), and atomQueuePut().

ATOM_TCB* queue_timer::tcb_ptr
Referenced by atomQueueGet(), and atomQueuePut().

sem_timer Struct Reference

Data Fields

ATOM_TCB * tcb_ptr
ATOM_SEM * sem_ptr

Field Documentation

ATOM_SEM* sem_timer::sem_ptr
Referenced by atomSemGet().

ATOM_TCB* sem_timer::tcb_ptr
Referenced by atomSemGet().

Cortex-M port

The Cortex-M port is different from the other architectures insofar as it makes use of two particular features. First, it uses separate stacks for thread and exception context. When the core enters exception mode, it first pushes xPSR, PC, LR, r0-r3 and r12 on the currently active stack (probably the thread stack in PSP) and then switches to the main stack stored in MSP. It also stores a special EXC_RETURN code in LR which, when loaded into the PC, will determine if on return program execution continues to use the MSP or switches over to the PSP and, on cores with an FPU, whether FPU registers need to be restored. The Cortex-M also implements a nested vectored interrupt controller (NVIC), which means that a running ISR may be pre-empted by an exception of higher priority.

The use of separate stacks for thread and exception context has the nice implication that you do not have to reserve space on every task's stack for possible use by ISRs. But it also means that it breaks atomthreads concept of simply swapping task stacks, regardless of if atomSched() was called from thread or interrupt context. We would have to implement different archContextSwitch() functions called from thread or exception context and also do messy stack manipulations depending on whether the task to be scheduled in was scheduled out in in the same context or not. Yuck! And don't get me started on nested exceptions calling atomIntExit()...

This is where the second feature comes handy, the PendSV mechanism. PendSV is an asynchronous exception with the lowest possible priority, which means that, when triggered, it will be called by the NVIC if there are no other exceptions pending or running. We use it by having archContextSwitch() set up a pointer to the TCB that should be scheduled in and then trigger the PendSv exception. As soon as program flow leaves the critical section or performs the outermost exception return, the pend_sv_handler() will be called and the thread context switch takes place.

atomport-private.h

```
#ifndef __ATOMPORT_PRIVATE_H_
#define __ATOMPORT_PRIVATE_H_

#include <stdint.h>
#include <stddef.h>
#include <libopencm3/cm3/cortex.h>
#include <stdlib.h>

/* Required number of system ticks per second (normally 100 for 10ms tick) */
#define SYSTEM_TICKS_PER_SEC            100

/* Size of each stack entry / stack alignment size (4 bytes on Cortex-M without
FPU) */
#define STACK_ALIGN_SIZE    sizeof(uint32_t)

#define ALIGN(x, a)         ((x + (__typeof__(x))(a) - 1) & ~((__typeof__(x))(a)
- 1))
#define PTR_ALIGN(p, a)     ((__typeof__(p))ALIGN((uint32_t)(p), (a)))
#define STACK_ALIGN(p, a)   (__typeof__(p))((__typeof__(a))(p) & ~((a) - 1))

#define POINTER             void *
#define UINT32              uint32_t

#define likely(x)           __builtin_expect(!!(x), 1)
#define unlikely(x)         __builtin_expect(!!(x), 0)
#define __maybe_unused      __attribute__((unused))

#define assert_static(e) \
   do { \
      enum { assert_static__ = 1/(e) }; \
   } while (0)

/**
 * Critical region protection: this should disable interrupts
 * to protect OS data structures during modification. It must
 * allow nested calls, which means that interrupts should only
 * be re-enabled when the outer CRITICAL_END() is reached.
 */
#define CRITICAL_STORE      bool __irq_flags
#define CRITICAL_START()    __irq_flags = cm_mask_interrupts(true)
#define CRITICAL_END()      (void) cm_mask_interrupts(__irq_flags)

/**
 * When using newlib, define port private field in atom_tcb to be a
 * struct _reent.
 */
#if defined(__NEWLIB__)
struct cortex_port_priv {
    struct _reent reent;
};

#define THREAD_PORT_PRIV    struct cortex_port_priv port_priv
#endif

/* Uncomment to enable stack-checking */
```

```
/* #define ATOM_STACK_CHECKING */

#endif /* __ATOM_PORT_H */
```

atomport-private.h

```
#ifndef __ATOMPORT_PRIVATE_H_
#define __ATOMPORT_PRIVATE_H_

#include "atomport.h"
#include "atom.h"

/**
 * context saved automagically by exception entry
 */
struct isr_stack {
        uint32_t r0;
        uint32_t r1;
        uint32_t r2;
        uint32_t r3;
        uint32_t r12;
        uint32_t lr;
        uint32_t pc;
        uint32_t psr;
} __attribute__((packed));

struct isr_fpu_stack {
        uint32_t s0;
        uint32_t s1;
        uint32_t s2;
        uint32_t s3;
        uint32_t s4;
        uint32_t s5;
        uint32_t s6;
        uint32_t s7;
        uint32_t s8;
        uint32_t s9;
        uint32_t s10;
        uint32_t s11;
        uint32_t s12;
        uint32_t s13;
        uint32_t s14;
        uint32_t s15;
        uint32_t fpscr;
} __attribute__((packed));

/**
 *  remaining context saved by task switch ISR
 */
struct task_stack {
        uint32_t r4;
        uint32_t r5;
        uint32_t r6;
        uint32_t r7;
        uint32_t r8;
        uint32_t r9;
        uint32_t r10;
        uint32_t r11;
        uint32_t exc_ret;
} __attribute__((packed));

struct task_fpu_stack {
```

```
        uint32_t s16;
        uint32_t s17;
        uint32_t s18;
        uint32_t s19;
        uint32_t s20;
        uint32_t s21;
        uint32_t s22;
        uint32_t s23;
        uint32_t s24;
        uint32_t s25;
        uint32_t s26;
        uint32_t s27;
        uint32_t s28;
        uint32_t s29;
        uint32_t s30;
        uint32_t s31;
} __attribute__((packed));

/**
 * Info needed by pend_sv_handler used for delayed task switching.
 * Running_tcb is a pointer to the TCB currently running (gosh, really?!)
 * next_tcb is a pointer to a TCB that should be running.
 * archContextSwitch() will update next_tcb and trigger a pend_sv. The
 * pend_sv_handler will be called as soon as all other ISRs have returned,
 * do the real context switch and update running_tcb.
 */
struct task_switch_info {
    volatile struct atom_tcb *running_tcb;
    volatile struct atom_tcb *next_tcb;
#if defined(__NEWLIB__)
    struct _reent *reent;
#endif
} __attribute__((packed));

#endif /* __ATOMPORT_PRIVATE_H_ */
```

atomport.c

```
#include <string.h>

#include <libopencm3/cm3/nvic.h>
#include <libopencm3/cm3/systick.h>
#include <libopencm3/cm3/cortex.h>
#include <libopencm3/cm3/scb.h>
#include <libopencm3/cm3/sync.h>

#include "atomport.h"
#include "atomport-private.h"
#include "asm_offsets.h"

static void thread_shell(void);

struct task_switch_info ctx_switch_info __asm__("CTX_SW_NFO") =
{
    .running_tcb = NULL,
    .next_tcb    = NULL,
};

extern void _archFirstThreadRestore(ATOM_TCB *);
void archFirstThreadRestore(ATOM_TCB *new_tcb_ptr)
{
#if defined(__NEWLIB__)
    ctx_switch_info.reent = &(new_tcb_ptr->port_priv.reent);
    __dmb();
#endif

    _archFirstThreadRestore(new_tcb_ptr);
}

/**
 * We do not perform the context switch directly. Instead we mark the new tcb
 * as should-be-running in ctx_switch_info and trigger a PendSv-interrupt.
 * The pend_sv_handler will be called when all other pending exceptions have
 * returned and perform the actual context switch.
 * This way we do not have to worry if we are being called from task or
 * interrupt context, which would mean messing with either main or thread
 * stack format.
 *
 * One difference to the other architectures is that execution flow will
 * actually continue in the old thread context until interrupts are enabled
 * again. From a thread context this should make no difference, as the context
 * switch will be performed as soon as the execution flow would return to the
 * calling thread. Unless, of course, the thread called atomSched() with
 * disabled interrupts, which it should not do anyways...
 */
void __attribute__((noinline))
archContextSwitch(ATOM_TCB *old_tcb_ptr __maybe_unused, ATOM_TCB *new_tcb_ptr)
{
    if(likely(ctx_switch_info.running_tcb != NULL)){
        ctx_switch_info.next_tcb = new_tcb_ptr;
#if defined(__NEWLIB__)
        ctx_switch_info.reent = &(new_tcb_ptr->port_priv.reent);
#endif
        __dmb();
```

```
        SCB_ICSR = SCB_ICSR_PENDSVSET;
    }
}

void sys_tick_handler(void)
{
    /* Call the interrupt entry routine */
    atomIntEnter();

    /* Call the OS system tick handler */
    atomTimerTick();

    /* Call the interrupt exit routine */
    atomIntExit(TRUE);
}

/**
 * Put chip into infinite loop if NMI or hard fault occurs
 */
void nmi_handler(void)
{
    while(1)
        ;
}

void hard_fault_handler(void)
{
    while(1)
        ;
}


/**
 * This function is called when a new thread is scheduled in for the first
 * time. It will simply call the threads entry point function.
 */
static void thread_shell(void)
{
    ATOM_TCB *task_ptr;

    /**
     * We "return" to here after being scheduled in by the pend_sv_handler.
     * We get a pointer to our TCB from atomCurrentContext()
     */
    task_ptr = atomCurrentContext();

    /**
     * Our thread entry point and parameter are stored in the TCB.
     * Call it if it is valid
     */
    if(task_ptr && task_ptr->entry_point){
        task_ptr->entry_point(task_ptr->entry_param);
    }

    /**
     * Thread returned or entry point was not valid.
     * Should never happen... Maybe we should switch MCU into debug mode here
     */
    while(1)
```

```
        ;
}

/**
 * Initialise a threads stack so it can be scheduled in by
 * archFirstThreadRestore or the pend_sv_handler.
 */
void archThreadContextInit(ATOM_TCB *tcb_ptr, void *stack_top,
                           void (*entry_point)(uint32_t), uint32_t entry_param)
{
    struct isr_stack *isr_ctx;
    struct task_stack *tsk_ctx;

    /**
     * Do compile time verification for offsets used in _archFirstThreadRestore
     * and pend_sv_handler. If compilation aborts here, you will have to adjust
     * the offsets for struct task_switch_info's members in asm-offsets.h
     */
    assert_static(offsetof(struct task_switch_info, running_tcb) ==
CTX_RUN_OFF);
    assert_static(offsetof(struct task_switch_info, next_tcb) == CTX_NEXT_OFF);
#if defined(__NEWLIB__)
    assert_static(offsetof(struct task_switch_info, reent) == CTX_REENT_OFF);
#endif

    /**
     * Enforce initial stack alignment
     */
    stack_top = STACK_ALIGN(stack_top, STACK_ALIGN_SIZE);

    /**
     * New threads will be scheduled from an exception handler, so we have to
     * set up an exception stack frame as well as task stack frame
     */
    isr_ctx = stack_top - sizeof(*isr_ctx);
    tsk_ctx = stack_top - sizeof(*isr_ctx) - sizeof(*tsk_ctx);

#if 0
    printf("[%s] tcb_ptr: %p stack_top: %p isr_ctx: %p tsk_ctx: %p entry_point:
%p, entry_param: 0x%x\n",
            __func__, tcb_ptr, stack_top, isr_ctx, tsk_ctx, entry_point,
entry_param);
    printf("[%s] isr_ctx->r0: %p isr_ctx->psr: %p tsk_ctx->r4: %p tsk_ctx->lr:
%p\n",
            __func__, &isr_ctx->r0, &isr_ctx->psr, &tsk_ctx->r4, &tsk_ctx->lr);
#endif
    /**
     * We use the exception return mechanism to jump to our thread_shell()
     * function and initialise the PSR to the default value (thumb state
     * flag set and nothing else)
     */
    isr_ctx->psr = 0x01000000;
    isr_ctx->pc  = (uint32_t) thread_shell;

    /* initialise unused registers to silly value */
    isr_ctx->lr  = 0xEEEEEEEE;
    isr_ctx->r12 = 0xCCCCCCCC;
    isr_ctx->r3  = 0x33333333;
    isr_ctx->r2  = 0x22222222;
    isr_ctx->r1  = 0x11111111;
```

```
    isr_ctx->r0  = 0x00000000;

    /**
     * We use this special EXC_RETURN code to switch from main stack to our
     * thread stack on exception return
     */
    tsk_ctx->exc_ret = 0xFFFFFFFD;

    /* initialise unused registers to silly value */
    tsk_ctx->r11 = 0xBBBBBBBB;
    tsk_ctx->r10 = 0xAAAAAAAA;
    tsk_ctx->r9  = 0x99999999;
    tsk_ctx->r8  = 0x88888888;
    tsk_ctx->r7  = 0x77777777;
    tsk_ctx->r6  = 0x66666666;
    tsk_ctx->r5  = 0x55555555;
    tsk_ctx->r4  = 0x44444444;

    /**
     * Stack frames have been initialised, save it to the TCB. Also set
     * the thread's real entry point and param, so the thread shell knows
     * what function to call.
     */
    tcb_ptr->sp_save_ptr = tsk_ctx;
    tcb_ptr->entry_point = entry_point;
    tcb_ptr->entry_param = entry_param;

#if defined(__NEWLIB__)
    /**
     * Initialise thread's reentry context for newlib
     */
    _REENT_INIT_PTR(&(tcb_ptr->port_priv.reent));
#endif
}
```

atomport-port.S

```
#include "asm_offsets.h"

.syntax unified

/**
 * Create more readable defines for usable intruction set and FPU
 */
#undef THUMB_2
#undef WITH_FPU

#if defined(__ARM_ARCH_7M__) || defined(__ARM_ARCH_7EM__)
#define THUMB_2
#endif

#if defined(__VFP_FP__) && !defined(__SOFTFP__)
#define WITH_FPU
#endif

/**
 * Extern variables needed for context switching and first thread restore
 */
.extern CTX_SW_NFO
.extern vector_table

#if defined(__NEWLIB__)
/**
 * When using newlib, reentry context needs to be updated on task switch
 */
.extern _impure_ptr
#endif

/**
 * Some bit masks and registers used
 */
.equ FPU_USED,      0x00000010
.equ SCB_ICSR,      0xE000ED04
.equ PENDSVCLR,     0x08000000

.text

.global _archFirstThreadRestore
.func   _archFirstThreadRestore
.type   _archFirstThreadRestore,%function
.thumb_func
_archFirstThreadRestore:
    /**
     * Disable interrupts. They should be disabled anyway, but just
     * to make sure...
     */
    movs    r1,         #1
    msr     PRIMASK,    r1

    /**
     * Reset main stack pointer to initial value, which is the first entry
     * in the vector table.
     */
```

```
    ldr     r1,         = vector_table
    ldr     r1,         [r1, #0]
    msr     MSP,        r1

    /* Update ctx_switch_info, set this thread as both running and next */
    ldr     r1,         = CTX_SW_NFO
    str     r0,         [r1, #CTX_RUN_OFF]
    str     r0,         [r1, #CTX_NEXT_OFF]

#if defined(__NEWLIB__)
    /**
     * Store the thread's reentry context address in _impure_ptr. This
     * will have been stored in ctx_switch_info.reent.
     */
    ldr     r2,         [r1, #CTX_REENT_OFF]
    ldr     r3,         = _impure_ptr
    str     r2,         [r3, #0]
#endif

    /* Get thread stack pointer from tcb. Conveniently the first element */
    ldr     r1,         [r0, #0]
    msr     PSP,        r1

    /**
     * Set bit #1 in CONTROL. Causes switch to PSP, so we can work directly
     * with SP now and use pop/push.
     */
    movs    r1,         #2
    mrs     r2,         CONTROL
    orrs    r2,         r2,     r1
    msr     CONTROL,    r2

    /**
     * Initialise thread's register context from its stack frame. Since this
     * function gets called only once at system start up, execution time is
     * not critical. We can get away with using only Thumb-1 instructions that
     * will work on all Cortex-M devices.
     *
     * Initial stack looks like this:
     * xPSR
     * PC
     * lr
     * r12
     * r3
     * r2
     * r1
     * r0
     * exc_ret <- ignored here
     * r11
     * r10
     * r9
     * r8
     * r7
     * r6
     * r5
     * r4 <- thread's saved_sp points here
     */

    /**
     *
```

```
     * Move SP to position of r8 and restore high registers by loading
     * them to r4-r7 before moving them to r8-r11
     */
    add     SP,         #16
    pop     {r4-r7}
    mov     r8,         r4
    mov     r9,         r5
    mov     r10,        r6
    mov     r11,        r7

    /* move SP back to top of stack and load r4-r7 */
    sub     SP,         #32
    pop     {r4-r7}

    /*load r12, lr, pc and xpsr to r0-r3 and restore r12 and lr */
    add     SP,         #36
    pop     {r0-r3}
    mov     r12,        r0
    mov     lr,         r1

    /**
     * r2 contains the PC and r3 APSR, SP is now at the bottom of the stack. We
     * can't initialise APSR now because we will have to do a movs later when
     * enabling interrupts, so r3 must not be touched. We also need an extra
     * register holding the value that will be moved to PRIMASK. To do this,
     * we build a new stack containing only the initial values of r2, r3
     * and pc. In the end this will be directly popped into the registers,
     * finishing the thread restore and branching to the thread's entry point.
     */

    /* Save PC value */
    push    {r2}

    /* Move values for r2 and r3 to lie directly below value for pc */
    sub     SP,         #20
    pop     {r1-r2}
    add     SP,         #12
    push    {r1-r2}

    /* Load values for r0 and r1 from stack */
    sub     SP,         #20
    pop     {r0-r1}

    /* Move SP to start of our new r2,r3,pc mini stack */
    add     SP,         #12

    /* Restore xPSR and enable interrupts */
    movs    r2,         #0
    msr     APSR_nzcvq, r3
    msr     PRIMASK,    r2

    /* Pop r2,r3,pc from stack, thereby jumping to thread entry point */
    pop     {r2,r3,pc}
    nop

.size   _archFirstThreadRestore, . - _archFirstThreadRestore
.endfunc

.global pend_sv_handler
.func   pend_sv_handler
```

```
.type   pend_sv_handler,%function
.thumb_func
pend_sv_handler:
    /**
     * Disable interrupts. No need to check if they were enabled because,
     * well, we're an interrupt handler. Duh...
     */
    movs    r0,         #1
    msr     PRIMASK,    r0

    /**
     * Clear PendSv pending bit. There seems to exist a hardware race condition
     * in the NVIC that can prevent automatic clearing of the PENDSVSET. See
     * http://embeddedgurus.com/state-space/2011/09/whats-the-state-of-your-
cortex/
     */
     ldr    r0,         = SCB_ICSR
     ldr    r1,         = PENDSVCLR
     str    r1,         [r0, #0]

    /**
     * Check if running and next thread are really different.
     * From here on we have
     * r0 = &ctx_switch_info
     * r1 = ctx_switch_info.running_tcb
     * r2 = ctx_switch_info.next_tcb
     *
     * If r1 == r2 we can skip the context switch. This may theoretically
     * happen if the running thread gets scheduled out and in again by
     * multiple nested or tail-chained ISRs before the PendSv handler
     * gets called.
     */
    ldr     r0,         = CTX_SW_NFO
    ldr     r1,         [r0, #CTX_RUN_OFF]
    ldr     r2,         [r0, #CTX_NEXT_OFF]
    cmp     r1,         r2
    beq     no_switch

    /**
     * Copy running thread's process stack pointer to r3 and use it to push
     * the thread's register context on its stack
     */
    mrs     r3,         PSP

#if defined(THUMB_2)
    /**
     * Save old thread's context on Cortex-M[34]
     */

#if defined(WITH_FPU)
    /* Check if FPU was used by thread and store registers if necessary */
    tst     lr,         FPU_USED
    it      eq
    vstmdbeq  r3!,      {s16-s31}

    /**
     * TODO: Defer stacking FPU context by disabling FPU and using a
     * fault handler to store the FPU registers if another thread
     * tries using it
     */
```

```
#endif // WITH_FPU

    /* Push running thread's remaining registers on stack */
    stmdb   r3!,        {r4-r11, lr}

#else // !THUMB2

    /**
     * Save old thread's register context on Cortex-M0.
     * Push running thread's remaining registers on stack.
     * Thumb-1 can use stm only on low registers, so we
     * have to do this in two steps.
     */

    /* Reserve space for r8-r11 + exc_return before storing r4-r7 */
    subs    r3,         r3,     #36
    stmia   r3!,        {r4-r7}

    /**
     * Move r8-r11 to low registers and use store multiple with automatic
     * post-increment to push them on the stack
     */
    mov     r4,         r8
    mov     r5,         r9
    mov     r6,         r10
    mov     r7,         r11
    stmia   r3!,        {r4-r7}

    /**
     * Move lr (contains the exc_return code) to low registers and store it
     * on the stack.
     */
    mov     r4,         lr
    str     r4,         [r3, #0]

    /* Re-adjust r3 to point at top of stack */
    subs    r3,         r3, #32
#endif // !THUMB_2
    /**
     * Address of running TCB still in r1. Store thread's current stack top
     * into its sp_save_ptr, which is the struct's first element.
     */
    str     r3,         [r1, #0]

    /**
     * ctx_switch_info.next_tcb is going to become ctx_switch_info.running_tcb,
     * so we update the pointer.
     */
    str     r2,         [r0, #CTX_RUN_OFF]

#if defined(__NEWLIB__)
    /**
     * Store the thread's reentry context address in _impure_ptr. This
     * will have been stored in ctx_switch_info.reent.
     */
    ldr     r4,         [r0, #CTX_REENT_OFF]
    ldr     r3,         = _impure_ptr
    str     r4,         [r3, #0]
#endif
```

```
    /**
     * Fetch next thread's stack pointer from its TCB's sp_save_ptr and restore
     * the thread's register context.
     */
    ldr     r3,         [r2, #0]

#if defined(THUMB_2)

    /* Cortex-M[34], restore thread's task stack frame */
    ldmia   r3!,        {r4-r11, lr}

#if defined(WITH_FPU)
    /**
     * Check if FPU was used by new thread and restore registers if necessary.
     */
    tst     lr,         FPU_USED
    it      eq
    vldmiaeq  r3!,      {s16-s31}

    /**
     * TODO: only restore FPU registers if FPU was used by another thread
     * between this thread being scheduled out and now.
     */
#endif // WITH_FPU
#else // !THUMB_2

    /**
     * Thread restore for Cortex-M0
     * Restore thread's task stack frame. Because thumb 1 only supports
     * load multiple on low register, we have to do it in two steps and
     * adjust the stack pointer manually.
     */

    /* Restore high registers */
    adds    r3,         r3, #16
    ldmia   r3!,        {r4-r7}
    mov     r8,         r4
    mov     r9,         r5
    mov     r10,        r6
    mov     r11,        r7

    /* Restore lr */
    ldr     r4,         [r3, #0]
    mov     lr,         r4
    subs    r3,         r3, #32

    /**
     * Restore r4-r7 and adjust r3 to point at the top of the exception
     * stack frame.
     */
    ldmia   r3!,        {r4-r7}
    adds    r3,         r3, #20
#endif // !THUMB_2

    /* Set process stack pointer to new thread's stack*/
    msr     PSP,        r3

no_switch:
    /* Re-enable interrupts */
    movs    r0,         #0
```

```
    msr     PRIMASK,    r0

    /* Return to new thread */
    bx      lr
    nop
.size   pend_sv_handler, . - pend_sv_handler
.endfunc
```