McRTOS

A Multi-core RTOS

Germán Rivera

McRTOS Features Overview

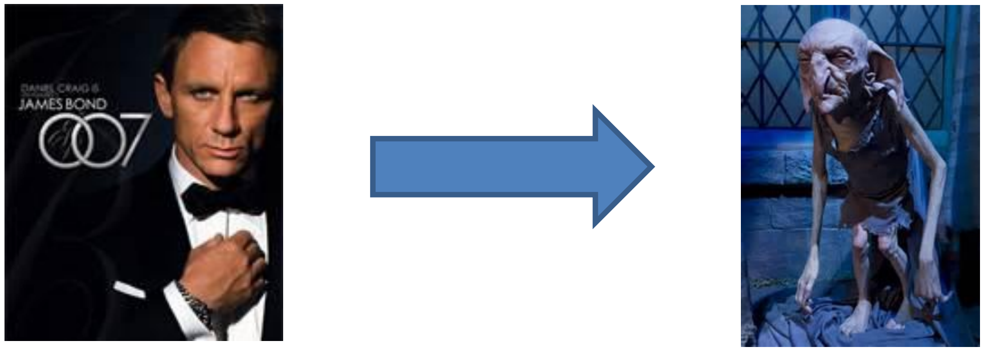
- Preemptive kernel
- Round-robin scheduler with priorities (and O(1) scheduler)
- Nested interrupts support
- Mutexes with priority inheritance (priority boosting) to prevent unbounded thread priority inversion
- Condition variables instead of semaphores
- Software timers use a timing wheel algorithm for efficient timer start, timer bookkeeping and timer stop.
- Multi-core support for “static” SMP
- Application threads run in unprivileged mode (ARM User mode) and McRTOS kernel runs in privileged mode (ARM System and IRQ modes).
- System call interface to protect access to McRTOS kernel services
- McRTOS APIs accessible to application threads via system calls. APIs return and receive pointers to opaque objects as parameters to prevent applications from tampering McRTOS internal data structures.
- First failure data capture (FFDC) support

McRTOS API Philosophy

- For the sake of determinism, static is better than dynamic:
  - Application thread architecture must be defined statically at compile-time
  - Threads should not be created/terminated dynamically
  - Threads should terminate only due an error
    - Thread creation API no exposed by default
    - Threads can be terminate themselves only by calling a "thread abort" API and
  - aborted threads cannot be recycled for creating new threads
  - Mutexes, condvar, timers and other McRTOS resources should not be created/destroyed dynamically
    - Resource creation APIs not exposed by default
    - No resource destruction APIs provided
  - Static SMP support:
    - Threads cannot migrate fro one core to another
- Mutexes and condvars can only be used to synchronize threads running in the same core (if needed, inter-core communication can still be done using inter-processor interrupts and shared memory)
- Opaque object pointers are returned and received by McRTOS APIs

Lesson of Humility

- Having a clean and elegant design and code does not mean you can claim victory early.
  - The nastiest bugs are not major design mistakes but subtle and often trivial low-level/assembly coding bugs, that can easily cripple the reliability of the entire software no matter how carefully thought the design and elegant the code may be.



Lesson of Humility (2)

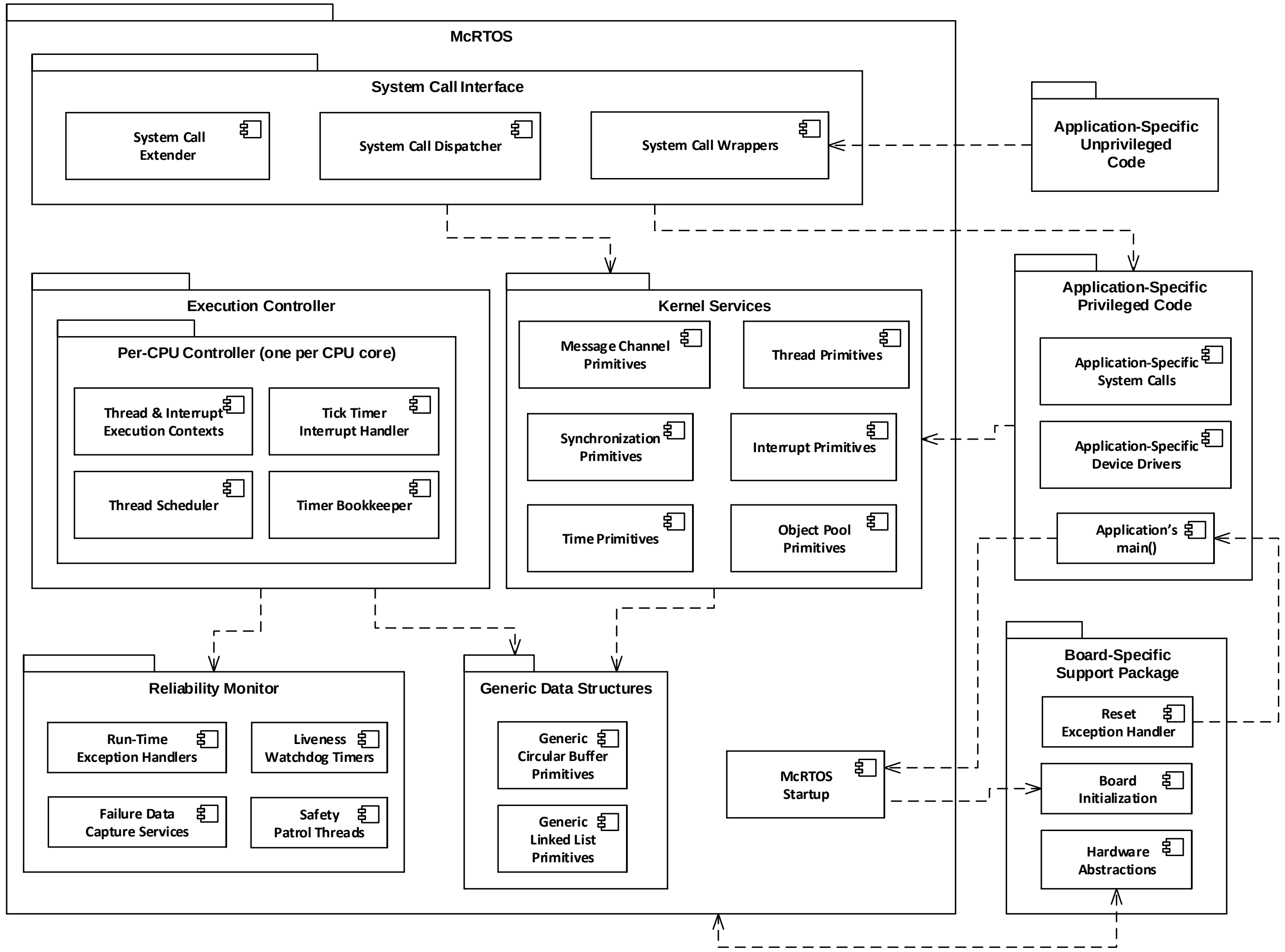
- These trivial almost insignificant bugs can cause a lot of debugging time, but ironically their fix is a one line change.
  - Some nasty bugs that caused my a lot of debugging time:
    - An "atomic increment" assembly routine for pointers to structures did not specify the size of the pointer, so only incremented one byte, instead of the structure size
    - interrupt stack too small
      - Imprinting the stack ands having "buffer zone" of at least one word at the
    - top end of the stack helped debug this
    - incorrect assembly instruction in context switch: ldmia r1, {r0-r12, lr, pc}^
      - Caught this thanks to the following assert:
        - » FDC_ASSERT_VALID_CODE_ADDRESS(cpu_lr_register);
        - » However this assert is wrong in general as the compiler may generate
      - Code to use LR as a scratch register in the middle of a routine
    - Blindly switching back to the interrupted CPU mode in an ISR to capture the interrupted context SP and LR. This is wrong if interrupted CPU mode is user mode, as there is no way to switch to IRQ mode from user mode by using msr (the nasty thing is that the processor does not generate an undefined instruction for this "msr" instruction, it just seems to ignore the instruction)

McRTOS Source Code Size

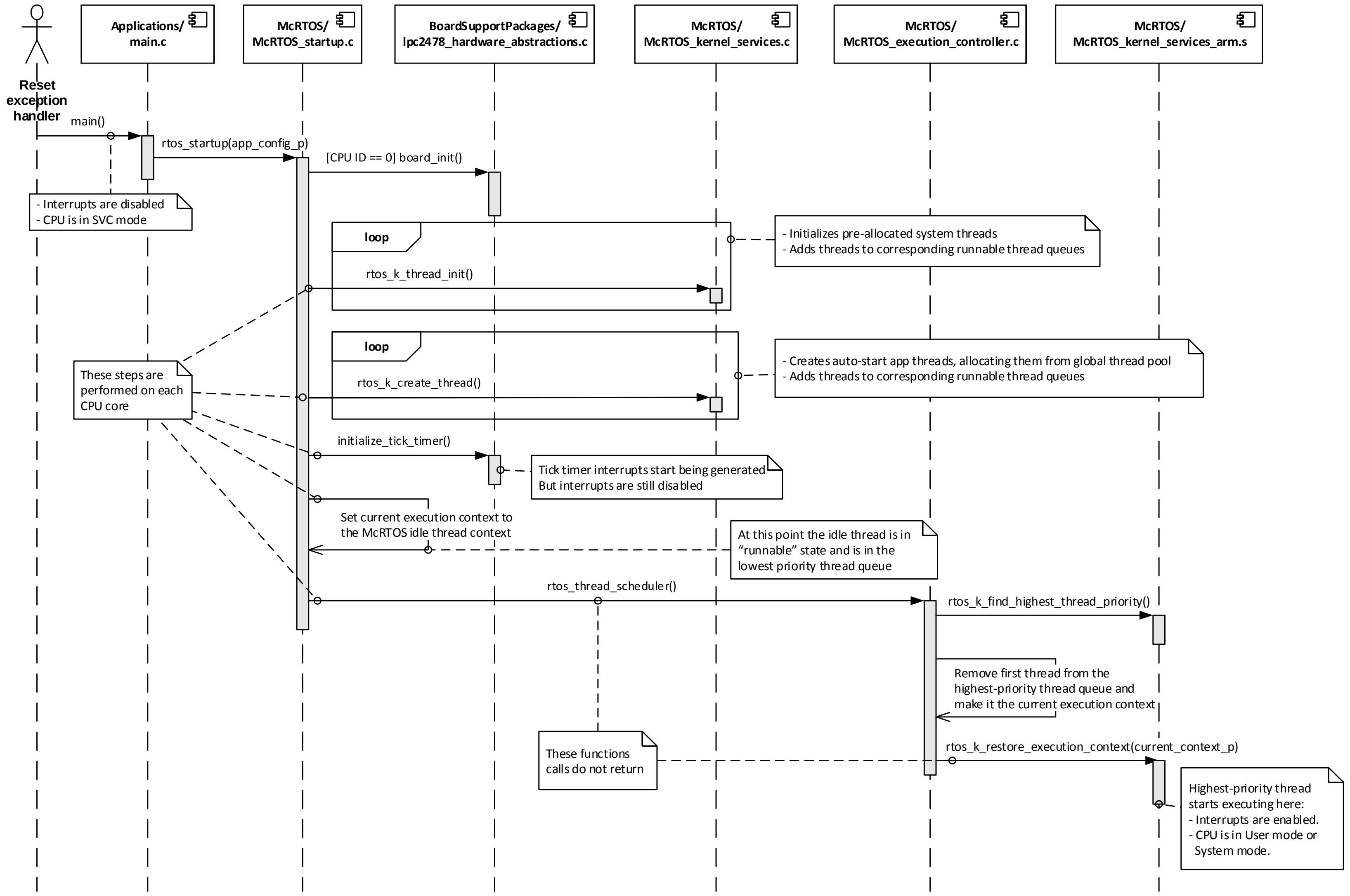
```
PS> wc -l .\src\McRTOS\*.s
  182 .\src\McRTOS\McRTOS_crt_armv4.s
  313
.\src\McRTOS\McRTOS_interrupt_service_routines_armv4.s
  635 .\src\McRTOS\McRTOS_kernel_services_armv4.s
  301 .\src\McRTOS\McRTOS_system_call_wrappers_armv4.s
  117 .\src\McRTOS\run_time_exception_handlers_armv4.s
 1548 total
PS> wc -l .\src\McRTOS\*.c
   863 .\src\McRTOS\failure_data_capture.c
   366 .\src\McRTOS\generic_list.c
   489 .\src\McRTOS\McRTOS_execution_controller.c
  2377 .\src\McRTOS\McRTOS_kernel_services.c
   875 .\src\McRTOS\McRTOS_startup.c
   533 .\src\McRTOS\utils.c
  5503 total
```

```
PS> wc -l .\inc\McRTOS\*.h
   215 .\inc\McRTOS\arm_defs.h
    94 .\inc\McRTOS\compile_time_checks.h
   695 .\inc\McRTOS\failure_data_capture.h
   283 .\inc\McRTOS\generic_list.h
   488 .\inc\McRTOS\McRTOS.h
   168 .\inc\McRTOS\McRTOS_config_parameters.h
   762 .\inc\McRTOS\McRTOS_internals.h
   891 .\inc\McRTOS\McRTOS_kernel_services.h
    16 .\inc\McRTOS\McRTOS_system_calls.h
   116 .\inc\McRTOS\utils.h
  3728 total
```

McRTOS High-level Architecture

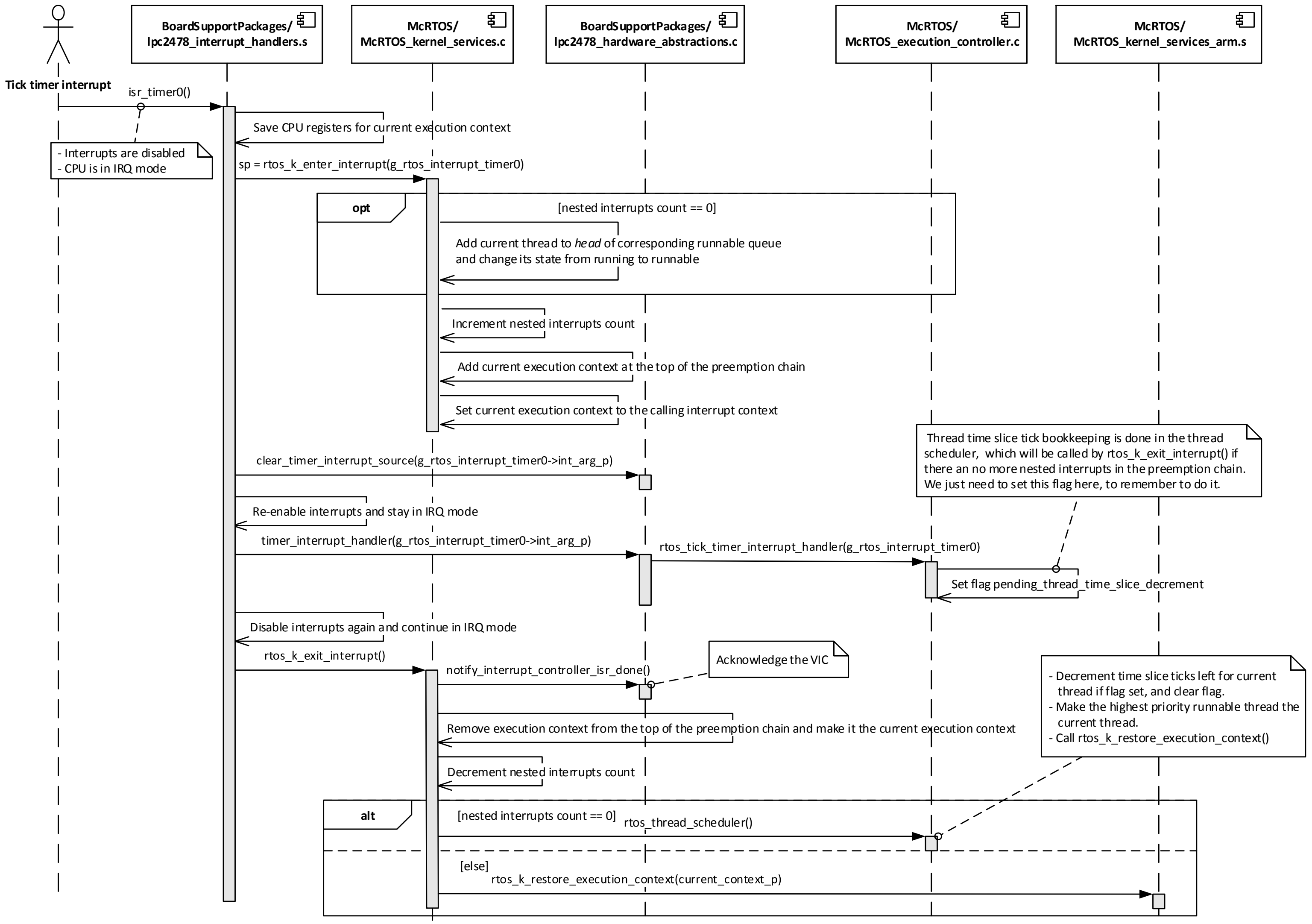

McRTOS
System Call Interface
System Call Extender
System Call Dispatcher
System Call Wrappers
Application-Specific Unprivileged Code
Execution Controller
Per-CPU Controller (one per CPU core)
Thread & Interrupt Execution Contexts
Tick Timer Interrupt Handler
Thread Scheduler
Timer Bookkeeper
Kernel Services
Message Channel Primitives
Thread Primitives
Synchronization Primitives
Interrupt Primitives
Time Primitives
Object Pool Primitives
Application-Specific Privileged Code
Application-Specific System Calls
Application-Specific Device Drivers
Application's main()
Reliability Monitor
Run-Time Exception Handlers
Liveness Watchdog Timers
Failure Data Capture Services
Safety Patrol Threads
Generic Data Structures
Generic Circular Buffer Primitives
Generic Linked List Primitives
McRTOS Startup
Board-Specific Support Package
Reset Exception Handler
Board Initialization
Hardware Abstractions

McRTOS Sartup Sequence



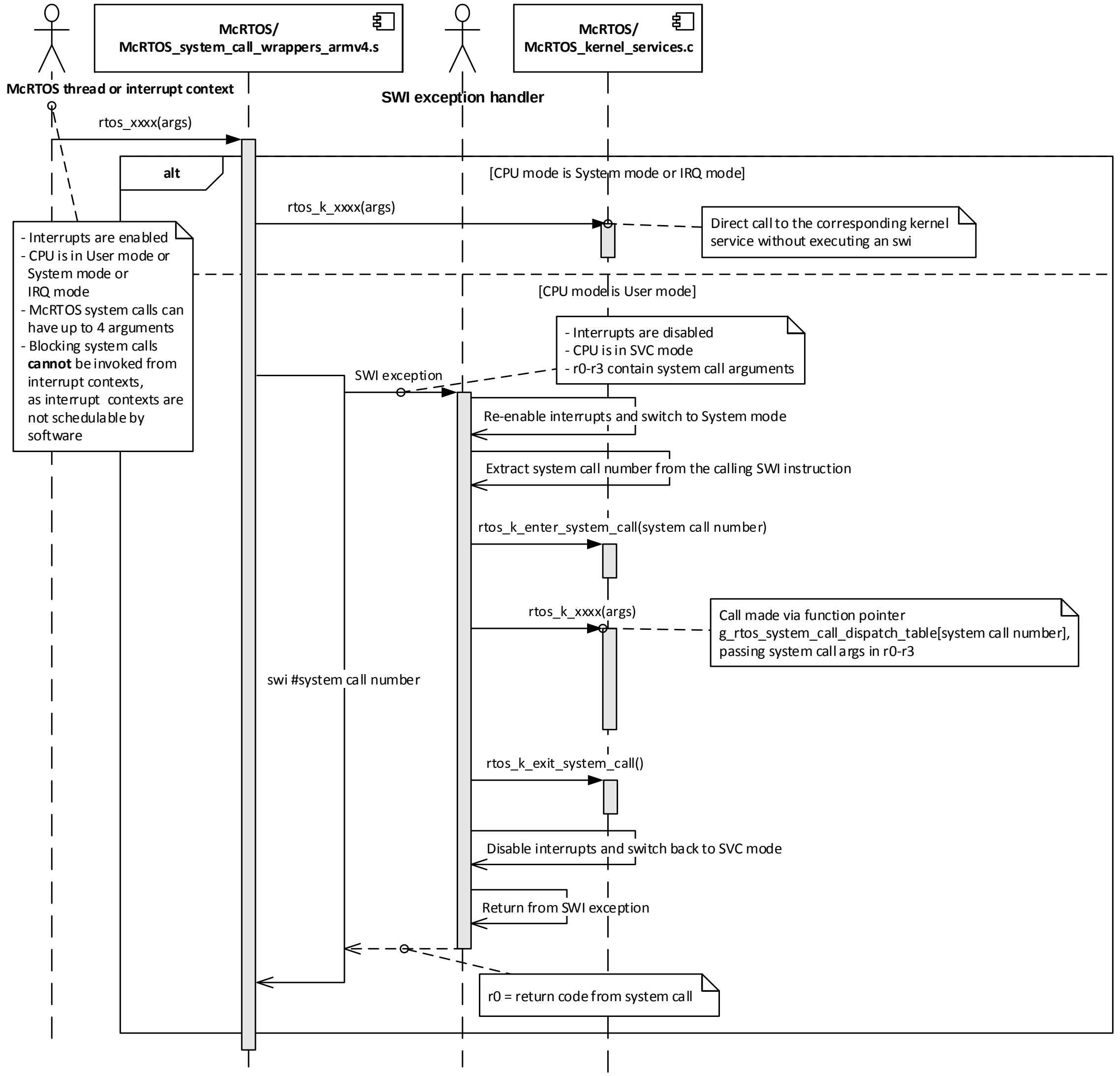
Reset exception handler
Applications/ main.c
McRTOS/ McRTOS_startup.c
BoardSupportPackages/ lpc2478_hardware_abstractions.c
McRTOS/ McRTOS_kernel_services.c
McRTOS/ McRTOS_execution_controller.c
McRTOS/ McRTOS_kernel_services_arm.s
main()
rtos_startup(app_config_p)
[CPU ID == 0] board_init()
- Interrupts are disabled
- CPU is in SVC mode
loop
rtos_k_thread_init()
- Initializes pre-allocated system threads
- Adds threads to corresponding runnable thread queues
loop
rtos_k_create_thread()
- Creates auto-start app threads, allocating them from global thread pool
- Adds threads to corresponding runnable thread queues
These steps are performed on each CPU core
initialize_tick_timer()
Tick timer interrupts start being generated
But interrupts are still disabled
Set current execution context to the McRTOS idle thread context
At this point the idle thread is in "runnable" state and is in the lowest priority thread queue
rtos_thread_scheduler()
rtos_k_find_highest_thread_priority()
Remove first thread from the highest-priority thread queue and make it the current execution context
rtos_k_restore_execution_context(current_context_p)
These functions calls do not return
Highest-priority thread starts executing here:
- Interrupts are enabled.
- CPU is in User mode or System mode.

McRTOS Tick Timer Interrupt

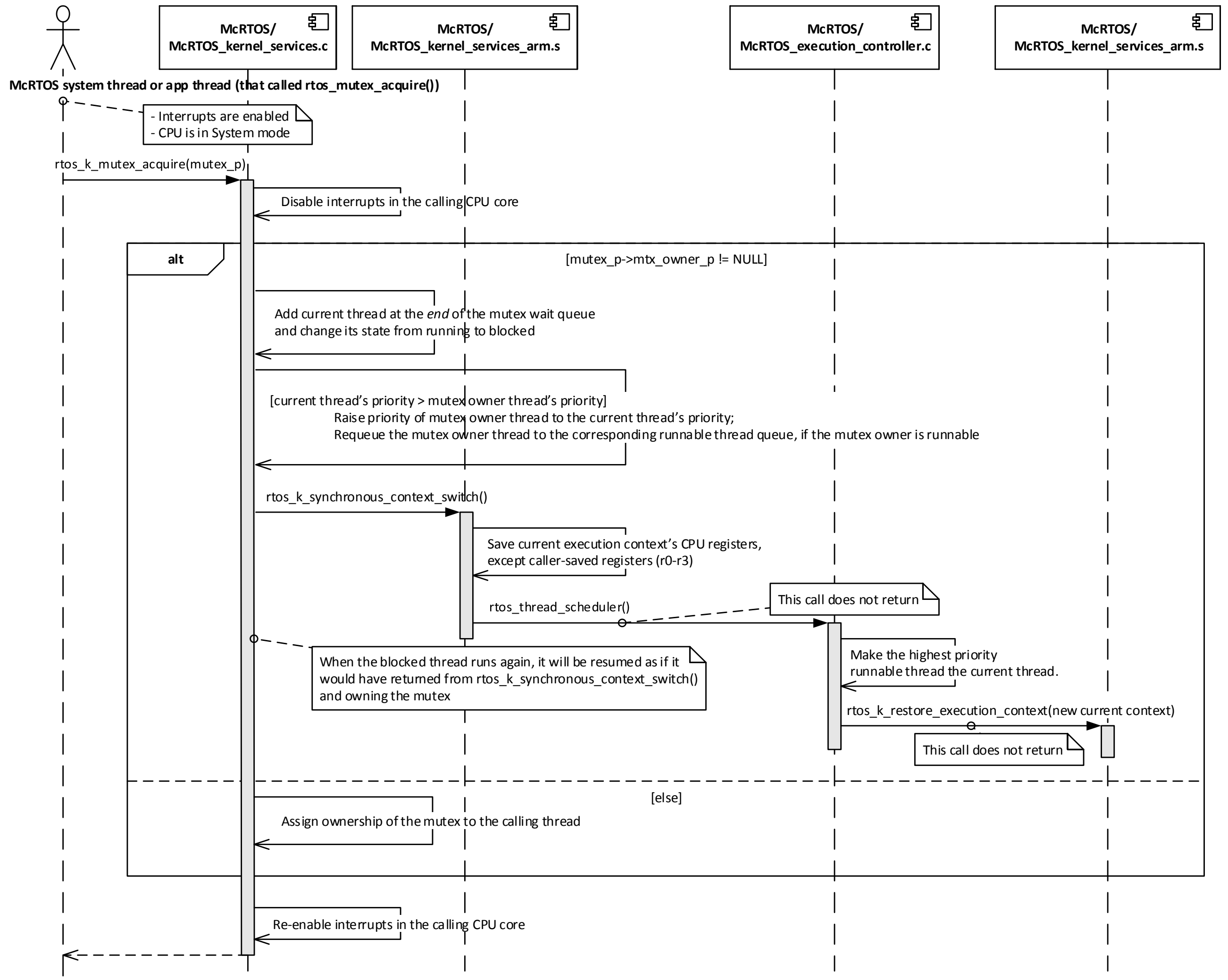


Tick timer interrupt
BoardSupportPackages/ lpc2478_interrupt_handlers.s
McRTOS/ McRTOS_kernel_services.c
BoardSupportPackages/ lpc2478_hardware_abstractions.c
McRTOS/ McRTOS_execution_controller.c
McRTOS/ McRTOS_kernel_services_arm.s
isr_timer0()
- Interrupts are disabled
- CPU is in IRQ mode
Save CPU registers for current execution context
sp = rtos_k_enter_interrupt(g_rtos_interrupt_timer0)
opt
[nested interrupts count == 0]
Add current thread to *head* of corresponding runnable queue and change its state from running to runnable
Increment nested interrupts count
Add current execution context at the top of the preemption chain
Set current execution context to the calling interrupt context
Thread time slice tick bookkeeping is done in the thread scheduler, which will be called by rtos_k_exit_interrupt() if there an no more nested interrupts in the preemption chain. We just need to set this flag here, to remember to do it.
clear_timer_interrupt_source(g_rtos_interrupt_timer0->int_arg_p)
Re-enable interrupts and stay in IRQ mode
timer_interrupt_handler(g_rtos_interrupt_timer0->int_arg_p)
rtos_tick_timer_interrupt_handler(g_rtos_interrupt_timer0)
Set flag pending_thread_time_slice_decrement
Disable interrupts again and continue in IRQ mode
rtos_k_exit_interrupt()
notify_interrupt_controller_isr_done()
Acknowledge the VIC
- Decrement time slice ticks left for current thread if flag set, and clear flag.
- Make the highest priority runnable thread the current thread.
- Call rtos_k_restore_execution_context()
Remove execution context from the top of the preemption chain and make it the current execution context
Decrement nested interrupts count
alt
[nested interrupts count == 0]
rtos_thread_scheduler()
[else]
rtos_k_restore_execution_context(current_context_p)

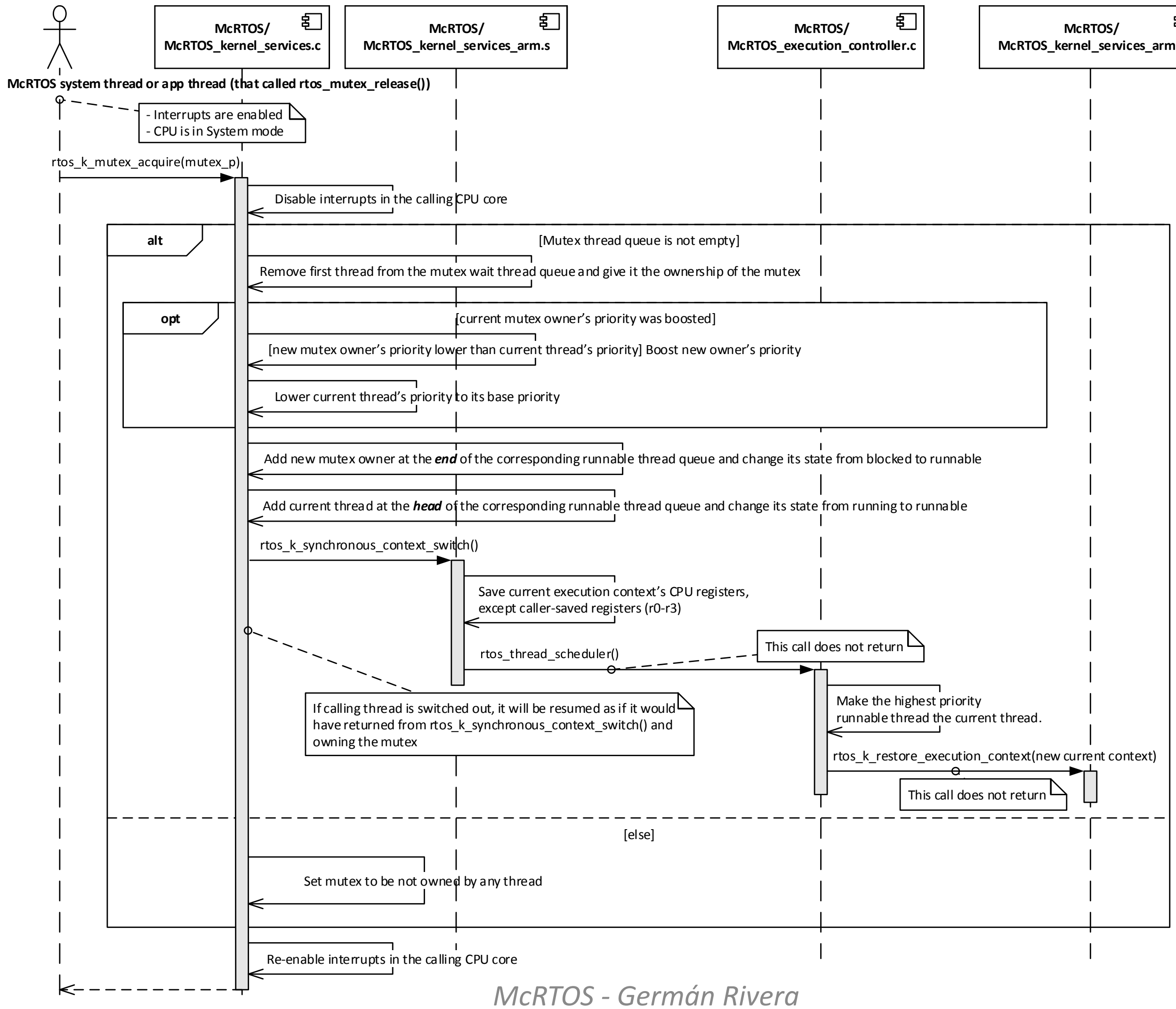
McRTOS System Call Sequence


McRTOS thread or interrupt context
McRTOS/ McRTOS_system_call_wrappers_armv4.s
SWI exception handler
McRTOS/ McRTOS_kernel_services.c
rtos_xxxx(args)
alt
[CPU mode is System mode or IRQ mode]
rtos_k_xxxx(args)
Direct call to the corresponding kernel service without executing an swi
- Interrupts are enabled
- CPU is in User mode or System mode or IRQ mode
- McRTOS system calls can have up to 4 arguments
- Blocking system calls **cannot** be invoked from interrupt contexts, as interrupt contexts are not schedulable by software
[CPU mode is User mode]
- Interrupts are disabled
- CPU is in SVC mode
- r0-r3 contain system call arguments
SWI exception
Re-enable interrupts and switch to System mode
Extract system call number from the calling SWI instruction
rtos_k_enter_system_call(system call number)
rtos_k_xxxx(args)
Call made via function pointer g_rtos_system_call_dispatch_table[system call number], passing system call args in r0-r3
swi #system call number
rtos_k_exit_system_call()
Disable interrupts and switch back to SVC mode
Return from SWI exception
r0 = return code from system call

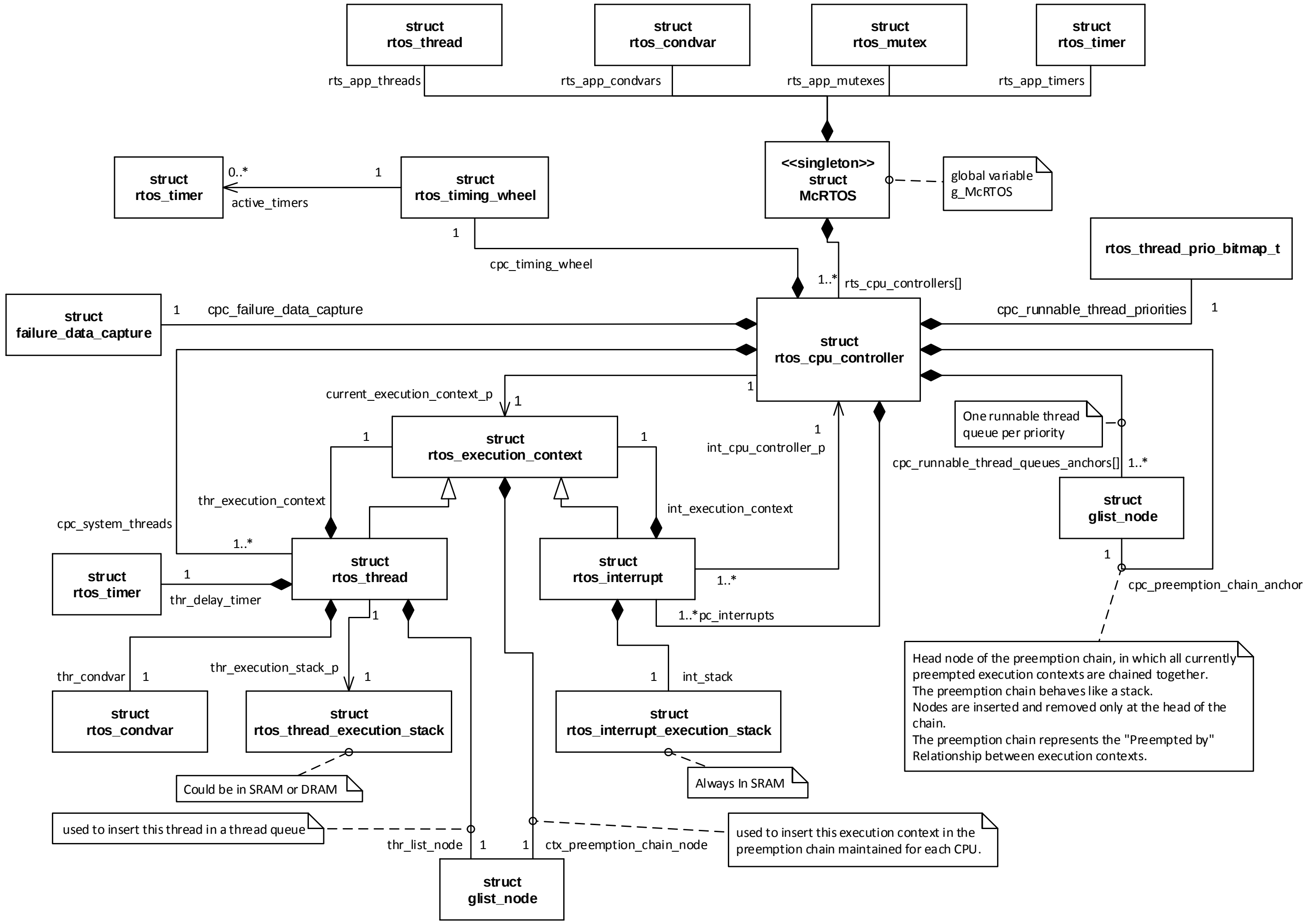
McRTOS Synchronous Context Switch when Acquiring a Mutex


McRTOS/ McRTOS_kernel_services.c
McRTOS/ McRTOS_kernel_services_arm.s
McRTOS/ McRTOS_execution_controller.c
McRTOS/ McRTOS_kernel_services_arm.s
McRTOS system thread or app thread (that called rtos_mutex_acquire())
- Interrupts are enabled
- CPU is in System mode
rtos_k_mutex_acquire(mutex_p)
Disable interrupts in the calling CPU core
alt
[mutex_p->mtx_owner_p != NULL]
Add current thread at the *end* of the mutex wait queue and change its state from running to blocked
[current thread's priority > mutex owner thread's priority]
Raise priority of mutex owner thread to the current thread's priority;
Requeue the mutex owner thread to the corresponding runnable thread queue, if the mutex owner is runnable
rtos_k_synchronous_context_switch()
Save current execution context's CPU registers, except caller-saved registers (r0-r3)
rtos_thread_scheduler()
This call does not return
Make the highest priority runnable thread the current thread.
When the blocked thread runs again, it will be resumed as if it would have returned from rtos_k_synchronous_context_switch() and owning the mutex
rtos_k_restore_execution_context(new current context)
This call does not return
[else]
Assign ownership of the mutex to the calling thread
Re-enable interrupts in the calling CPU core

McRTOS Synchronous Context Switch when Releasing a Mutex

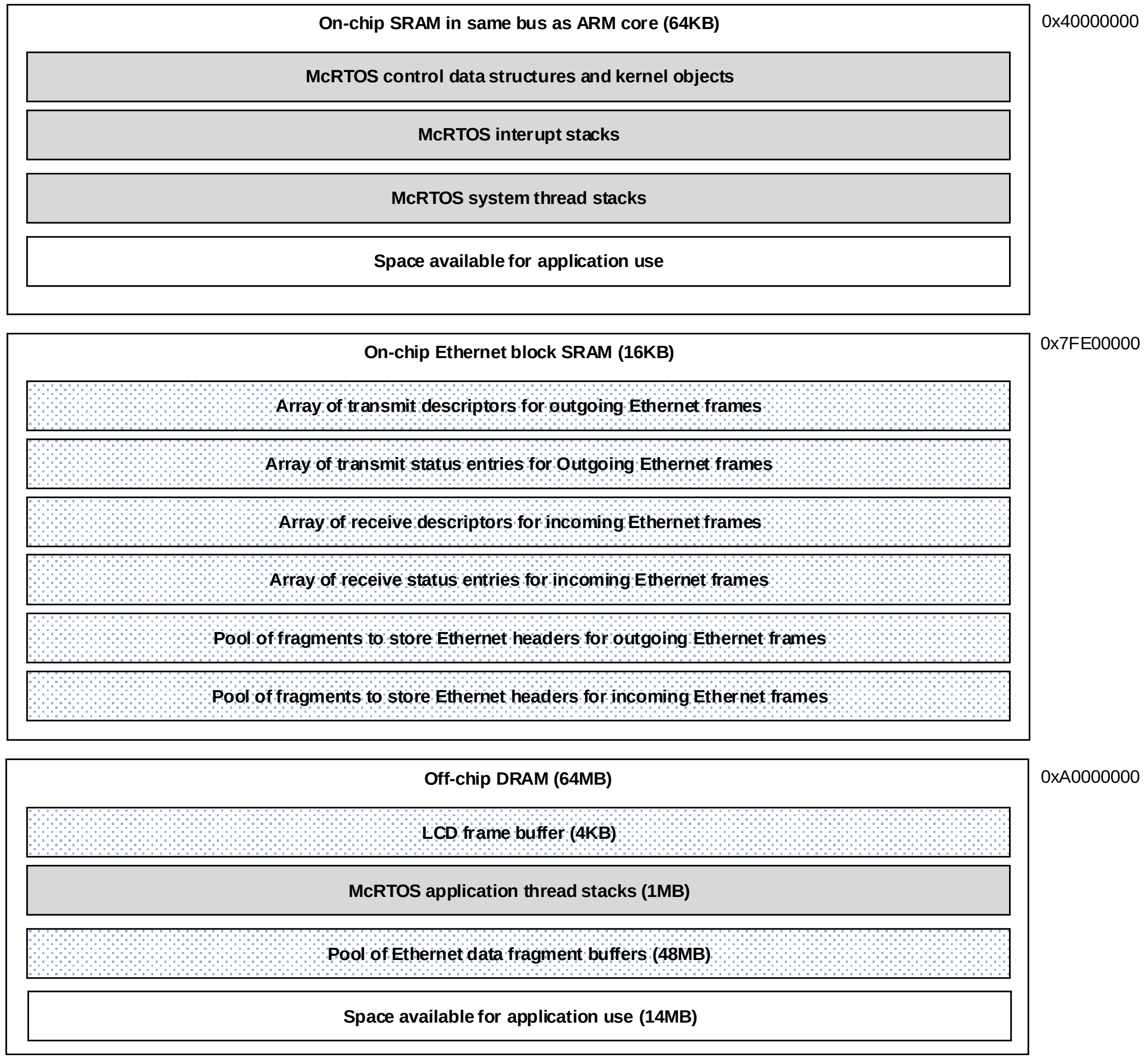

McRTOS/ McRTOS_kernel_services.c
McRTOS/ McRTOS_kernel_services_arm.s
McRTOS/ McRTOS_execution_controller.c
McRTOS/ McRTOS_kernel_services_arm.s
McRTOS system thread or app thread (that called rtos_mutex_release())
- Interrupts are enabled
- CPU is in System mode
rtos_k_mutex_acquire(mutex_p)
Disable interrupts in the calling CPU core
alt
[Mutex thread queue is not empty]
Remove first thread from the mutex wait thread queue and give it the ownership of the mutex
opt
[current mutex owner's priority was boosted]
[new mutex owner's priority lower than current thread's priority] Boost new owner's priority
Lower current thread's priority to its base priority
Add new mutex owner at the end of the corresponding runnable thread queue and change its state from blocked to runnable
Add current thread at the head of the corresponding runnable thread queue and change its state from running to runnable
rtos_k_synchronous_context_switch()
Save current execution context's CPU registers, except caller-saved registers (r0-r3)
rtos_thread_scheduler()
This call does not return
If calling thread is switched out, it will be resumed as if it would have returned from rtos_k_synchronous_context_switch() and owning the mutex
Make the highest priority runnable thread the current thread.
rtos_k_restore_execution_context(new current context)
This call does not return
[else]
Set mutex to be not owned by any thread
Re-enable interrupts in the calling CPU core

McRTOS Major Data Structures



struct rtos_thread
struct rtos_condvar
struct rtos_mutex
struct rtos_timer
rts_app_threads
rts_app_condvars
rts_app_mutexes
rts_app_timers
<<singleton>> struct McRTOS
global variable g_McRTOS
struct rtos_timer
0..*
active_timers
1
struct rtos_timing_wheel
1
cpc_timing_wheel
1..*
rts_cpu_controllers[]
rtos_thread_prio_bitmap_t
struct failure_data_capture
1
cpc_failure_data_capture
cpc_runnable_thread_priorities
1
struct rtos_cpu_controller
current_execution_context_p
1
1
One runnable thread queue per priority
struct rtos_execution_context
1
1
1
int_cpu_controller_p
cpc_runnable_thread_queues_anchors[]
1..*
thr_execution_context
int_execution_context
struct glist_node
cpc_system_threads
1..*
struct rtos_thread
struct rtos_interrupt
1
cpc_preemption_chain_anchor
struct rtos_timer
1
thr_delay_timer
1..*
1..*pc_interrupts
1
Head node of the preemption chain, in which all currently preempted execution contexts are chained together. The preemption chain behaves like a stack. Nodes are inserted and removed only at the head of the chain. The preemption chain represents the "Preempted by" Relationship between execution contexts.
thr_condvar
1
thr_execution_stack_p
1
1
int_stack
struct rtos_condvar
struct rtos_thread_execution_stack
struct rtos_interrupt_execution_stack
Could be in SRAM or DRAM
Always In SRAM
used to insert this thread in a thread queue
thr_list_node
1
1
ctx_preemption_chain_node
used to insert this execution context in the preemption chain maintained for each CPU.
struct glist_node

McRTOS LPC2478-STK RAM Map


On-chip SRAM in same bus as ARM core (64KB)
0x40000000
McRTOS control data structures and kernel objects
McRTOS interupt stacks
McRTOS system thread stacks
Space available for application use
On-chip Ethernet block SRAM (16KB)
0x7FE00000
Array of transmit descriptors for outgoing Ethernet frames
Array of transmit status entries for Outgoing Ethernet frames
Array of receive descriptors for incoming Ethernet frames
Array of receive status entries for incoming Ethernet frames
Pool of fragments to store Ethernet headers for outgoing Ethernet frames
Pool of fragments to store Ethernet headers for incoming Ethernet frames
Off-chip DRAM (64MB)
0xA0000000
LCD frame buffer (4KB)
McRTOS application thread stacks (1MB)
Pool of Ethernet data fragment buffers (48MB)
Space available for application use (14MB)