MAKE with Ada

Programming Competition May 15 - September 15, 2017

Design of the McRTOS Multi-core Real-Time Operating System

Germán Rivera
jgrivera67@gmail.com

August 5, 2017

Abstract

This document describes the design of McRTOS, a Multi-core Real-time Operating System kernel for ARM processors [1, 2]. Specifically, McRTOS is designed to support ARM cores that implement the ARMv7 architecture [3] in any of its profiles (A, R or M), and also to be backward compatible down to the ARMv4 architecture. McRTOS is intended to support both single-core processors and multi-core processors.

The main focus of McRTOS is reliability. McRTOS is designed to be thoroughly self-checking and it is intended to be used in safety-critical applications.

The design of McRTOS is described as a formal specification using the Z notation [5, 6]. Z is a software modeling notation based on discrete mathematics structures (such as sets, relations and functions) and predicate logic. With Z, data structures can be specified in terms of mathematical structures and their state invariants can be specified using mathematical predicates. The pre-conditions and post-conditions of the operations that manipulate the data structures can also be specified using predicates. Using Z for this purpose encourages a rigorous and methodical thought process to elicit the correctness properties to check at run time, in a systematic way. The Z specification described here was checked with the `fuzz` tool [7], a Z type-checker, that catches Z type mismatches in predicates.

Contents

List of Figures

Chapter 1

Overview

1.1 Major Design Decisions and Assumptions

- Threads will always run in the CPU core in which they were created. They cannot migrate to a different core. Mutexes and condition variables can only be used to synchronize threads in the same CPU core. For synchronization/-communication between threads on separate CPU cores, a door-bell mechanism will be implemented using shared memory and software triggered inter-processor interrupts.
- An application thread can only create other threads, mutexes, condition variables and timers on the same CPU core in which it runs.

1.2 McRTOS Component Architecture

The McRTOS component-level architecture is depicted in the UML component diagram in figure 1.1.

Figure 1.1: McRTOS Component Architecture

1.3 Development Plan

McRTOS will be developed initially for the following two embedded boards:

- The Olimex LPC2478-STK development prototype board. This board features an LPC2478 SoC, which has an ARM7TDMI CPU core, 64K of static RAM and 512K of Flash. The ARM7TDMI core implements the ARMv4 archirecture.
- The Stellaris LM4F120 LaunchPad evaluation board. This board features an LM4F120H5QR SoC, which has an ARM Cortex-M4 core, 32K of static RAM and 256K of Flash. The Cortex-M4 core implements the ARM v7-M architecture.

Although the SoCs on these boards have single core CPUs, McRTOS will be designed from the beginning with support of Multi-core CPUs.

The development of McRTOS will be done in the following phases:

- Phase 1: Design, code and test the following components:
  - Source tree architecture and makefiles
  - Minimal board support for LPC2478-STK
  - Failure Data Capture Services
  - Runtime Exception Handlers
  - Generic Linked-List Primitives
  - Thread Primitives: Thread Queue
- Phase 2: Design, code and test the following components:
  - Thread Primitives: thread creation
  - Tick Timer Interrupt Handler and Context Switch for the LPC2478 SoC
  - Thread Scheduler
  - Minimal board support for LM4F120
  - Tick Timer Interrupt Handler and Context Switch for the LM4F120 SoC
- Phase 3: Design, code and test the following components:
  - Synchronization Primitives
  - Time Primitives: Delay
  - Generic Circular Buffer Primitives
  - Object Pool Primitives

  - Message Channel Primitives

- Phase 4: Design, code and test the following components:
  - Interrupt Primitives for Application-specific Peripheral interrupts
  - Time Primitives: timers
  - System Call Interface components
  - Liveness Watchdog Timers
  - Safety Patrol Threads

- Phase 5: Design, code and test the following components:
  - Peripheral device drivers for the LPC2478-STK board
  - Peripheral device drivers for the LM4F120 LaunchPad board

Chapter 2

Major Data Structures

2.1 Z Naming Conventions

The following naming conventions are used in this Z specification.

- Z Primitive types are in uppercase.
- Z Composite types (schema types) start with uppercase.
- Z constants and variables start with lower case, with the following special cases:
  - Identifiers that start with k represent compile-time constants.
  - Identifiers that start with z are redundant variables that are only used to add clarity to the specification. They are not meant to be implemented in code.

2.2 Numeric Constants

These are compile-time constants. Values given are just for illustrative purposes. Some may vary in the actual implementation.

$$
\begin{array}{|l}
\mathit{kMaxNumCpuCores} : \mathbb{N}_1 \\
\mathit{kMaxNumThreadsPerCpu} : \mathbb{N}_1 \\
\mathit{kMaxNumMutexes} : \mathbb{N}_1 \\
\mathit{kMaxNumCondvars} : \mathbb{N}_1 \\
\mathit{kWordSizeInBytes} : \mathbb{N}_1 \\
\mathit{kMaxNumInterruptChannelsPerCpu} : \mathbb{N}_1 \\
\mathit{kMaxNumThreadPriorities} : \mathbb{N}_1 \\
\mathit{kThreadQuantumInTicks} : \mathbb{N}_1 \\
\mathit{kLowestThreadPriority} : \mathbb{N}_1 \\
\mathit{kThreadStackSizeInWords} : \mathbb{N}_1 \\
\mathit{kInterruptStackSizeInWords} : \mathbb{N}_1 \\
\hline
\mathit{kMaxNumCpuCores} = 4 \\
\mathit{kMaxNumThreadsPerCpu} = 32 \\
\mathit{kMaxNumMutexes} = 16 \\
\mathit{kMaxNumCondvars} = 16 \\
\mathit{kWordSizeInBytes} = 4 \\
\mathit{kMaxNumInterruptChannelsPerCpu} = 32 \\
\mathit{kMaxNumThreadPriorities} = 32 \\
\mathit{kLowestThreadPriority} = \mathit{kMaxNumThreadPriorities} - 1 \\
\mathit{kThreadQuantumInTicks} = 8 \\
\mathit{kThreadStackSizeInWords} = 128 \\
\mathit{kInterruptStackSizeInWords} = 32
\end{array}
$$

kMaxNumThreadPriorities is limited to the integer bit-size of the machine (32-bits for ARMv7). This limitation is to ensure that the thread scheduler has "O(1)" complexity (requiring only one `CLZ` ARM instruction to determine the highest-priority runnable thread).

2.3 Primitive Types

$$
\begin{array}{l}
[\mathit{BYTE\_LOCATION}] \\
[\mathit{WORD\_LOCATION}] \\
[\mathit{OBJECT\_TYPE}] \\
[\mathit{LIST\_NODE}] \\
[\mathit{STRING}] \\
\mathit{CPUID} == 0 \mathrel{..} \mathit{kMaxNumCpuCores} - 1 \\
\mathit{THREAD\_ID} == 0 \mathrel{..} \mathit{kMaxNumThreadsPerCpu} - 1 \\
\mathit{INTERRUPT\_CHANNEL} == 0 \mathrel{..} \mathit{kMaxNumInterruptChannelsPerCpu} - 1 \\
\mathit{THREAD\_PRIO} == 0 \mathrel{..} \mathit{kMaxNumThreadPriorities} - 1 \\
\mathit{INTPRIO} == 0 \mathrel{..} \mathit{kMaxNumInterruptChannelsPerCpu} - 1 \\
\mathit{UINT8} == 0 \mathrel{..} 255 \\
\mathit{UINT16} == 0 \mathrel{..} 65535 \\
\mathit{UINT32} == 0 \mathrel{..} 4294967295 \\
\mathit{ADDRESS} == \mathit{UINT32} \\
\mathit{EXECUTION\_CONTEXT\_TYPE} ::= \mathit{kThreadContext} \mid \mathit{kInterruptContext} \\
\mathit{CPU\_MODE} ::= \mathit{kUnprivilegedThreadMode} \mid \\
\qquad\qquad\qquad \mathit{kPrivilegedThreadMode} \mid \\
\qquad\qquad\qquad \mathit{kInterruptMode} \\
\mathit{SYNCHRONIZATION\_SCOPE} ::= \mathit{kLocalCpuThreadOnly} \mid \\
\qquad\qquad\qquad\qquad\qquad\qquad \mathit{kLocalCpuInterruptAndThread} \mid \\
\qquad\qquad\qquad\qquad\qquad\qquad \mathit{kInterCpuThreadOnly} \\
\mathit{THREAD\_STATE} ::= \mathit{kRunnable} \mid \\
\qquad\qquad\qquad\quad \mathit{kRunning} \mid \\
\qquad\qquad\qquad\quad \mathit{kBlocked} \mid \\
\qquad\qquad\qquad\quad \mathit{kPreemptedByInterrupt} \mid \\
\qquad\qquad\qquad\quad \mathit{kPreemptedByThread} \mid \\
\qquad\qquad\qquad\quad \mathit{kTerminated}
\end{array}
$$

For both threads and interrupts, lower priority numbers represent higher priorities. Interrupts have higher priority than threads. That is, the lowest priority interrupt has higher priority than the highest priority thread.

2.4 Structural Constants

$$
\begin{array}{|l}
kValidMemoryBytes : ADDRESS \rightarrowtail\!\!\!\!\!\!\!\;\; \mapsto BYTE\_LOCATION \\
kByteValue : BYTE\_LOCATION \rightarrow UINT8 \\
kValidMemoryWords : ADDRESS \rightarrowtail\!\!\!\!\!\!\!\;\; \mapsto WORD\_LOCATION \\
kWordValue : WORD\_LOCATION \rightarrow UINT32 \\
kValidRamWordAddresses : \mathbb{F}_1\, ADDRESS \\
kValidRomWordAddresses : \mathbb{F}\, ADDRESS \\
kValidMmioWordAddresses : \mathbb{F}\, ADDRESS \\
kReadOnlyAddresses : \mathbb{F}\, ADDRESS \\
kExecutableAddresses : \mathbb{F}\, ADDRESS \\
kInterruptPriority : INTERRUPT\_CHANNEL \rightarrow INTPRIO \\
\hline
\operatorname{dom} kValidMemoryBytes \subseteq \{a : ADDRESS\} \\
\operatorname{dom} kValidMemoryWords \subseteq \{a : ADDRESS \mid a \mathbin{\mathsf{mod}} kWordSizeInBytes = 0\} \\
\bigcap\{kValidRamWordAddresses, kValidRomWordAddresses, \\
\quad kValidMmioWordAddresses\} = \emptyset \\
\bigcup\{kValidRamWordAddresses, kValidRomWordAddresses, \\
\quad kValidMmioWordAddresses\} \subseteq \operatorname{dom} kValidMemoryWords \\
kValidRomWordAddresses \subseteq kReadOnlyAddresses \\
kReadOnlyAddresses \subset (kValidRomWordAddresses \cup kValidRamWordAddresses) \\
kExecutableAddresses \subset kReadOnlyAddresses
\end{array}
$$

2.5 State Variables

$$
\begin{array}{|l}
\multicolumn{1}{l}{\mathit{McRTOS}} \\
\hline
\mathit{ExecutionController} \\
\mathit{ReliabilityMonitor} \\
\mathit{mutexes} : \mathbb{F}\, \mathit{Mutex} \\
\mathit{condvars} : \mathbb{F}\, \mathit{Condvar} \\
\mathit{messageChannels} : \mathbb{F}\, \mathit{MessageChannel} \\
\mathit{objectPools} : \mathbb{F}\, \mathit{ObjectPool} \\
\mathit{zMutexOwner} : \mathit{Mutex} \pfun \mathit{ExecutionContext} \\
\mathit{zMutexWaiters} : \mathit{Thread} \pfun \mathit{Mutex} \\
\mathit{zCondvarWaiters} : \mathit{Thread} \pfun \mathit{Condvar} \\
\mathit{zCondvarToMutex} : \mathit{Condvar} \pfun \mathit{Mutex} \\
\hline
\operatorname{dom} \mathit{zMutexOwner} \subseteq \mathit{mutexes} \\[1ex]
\operatorname{ran} \mathit{zMutexOwner} \subset \\
\quad \{ t : \mathit{zThreads} \bullet t.\mathit{executionContext} \} \cup \{ i : \mathit{zInterrupts} \bullet i.\mathit{executionContext} \} \\[1ex]
\operatorname{dom} \mathit{zMutexWaiters} \subset \mathit{zThreads} \land \operatorname{ran} \mathit{zMutexWaiters} \subseteq \mathit{mutexes} \\[1ex]
\operatorname{dom} \mathit{zCondvarWaiters} \subset \mathit{zThreads} \land \operatorname{ran} \mathit{zCondvarWaiters} \subseteq \mathit{condvars} \\[1ex]
\operatorname{dom} \mathit{zMutexWaiters} \cap \operatorname{dom} \mathit{zCondvarWaiters} = \emptyset \\[1ex]
\# \mathit{zMutexWaiters} < \# \mathit{zThreads} \land \# \mathit{zCondvarWaiters} < \# \mathit{zThreads} \\[1ex]
\forall \mathit{cpu} : \mathit{CPUID} \bullet \\
\quad (\mathit{zCpuIdToCpuController}(\mathit{cpu})).\mathit{zRunnableThreads} \cap \\
\quad (\operatorname{dom} \mathit{zMutexWaiters} \cup \operatorname{dom} \mathit{zCondvarWaiters}) = \emptyset \\[1ex]
\forall t : \mathit{zThreads} \bullet \\
\quad t.\mathit{executionContext} \notin \operatorname{ran} \mathit{zMutexOwner} \Rightarrow \\
\qquad t.\mathit{currentPriority} = t.\mathit{basePriority} \\[1ex]
\{ \mathit{mq} : \mathit{messageChannels} \bullet \mathit{mq}.\mathit{mutex} \} \subset \mathit{mutexes} \\[1ex]
\{ \mathit{mq} : \mathit{messageChannels} \bullet \mathit{mq}.\mathit{notEmptyCondvar} \} \cup \\
\{ \mathit{mq} : \mathit{messageChannels} \bullet \mathit{mq}.\mathit{notFullCondvar} \} \subset \mathit{condvars} \\[1ex]
\forall \mathit{cv} : \operatorname{dom} \mathit{zCondvarToMutex} \bullet \\
\quad \mathit{cv}.\mathit{synchronizationScope} \neq \mathit{kLocalCpuInterruptAndThread} \land \\
\quad \mathit{cv}.\mathit{synchronizationScope} = (\mathit{zCondvarToMutex}(\mathit{cv})).\mathit{synchronizationScope} \\
\hline
\end{array}
$$

Invariants:

- The same thread cannot be waiting for more than one mutex.
- The same mutex cannot be owned by more than one thread.

- The same thread cannot be waiting on more than one condition variable.
- The same thread cannot be waiting on a mutex and a condition variable at the same time.
- The same thread cannot be both runnable and blocked on a condvar or mutex.
- ISRs can never wait on mutexes or condvars. However, ISRs can signal condvars for which waiting threads call "wait for interrupt".
- The current priority of a thread that does not own a mutex must always be its base priority.

2.5.1 Execution Controller

$$
\begin{array}{|l}
\textit{ExecutionController} \\
\hline
\textit{zCpuIdToCpuController} : \textit{CPUID} \rightarrowtail \textit{CpuController} \\
\textit{cpuControllers} : \mathbb{F}_1\ \textit{CpuController} \\
\textit{zExecutionContexts} : \mathbb{F}_1\ \textit{ExecutionContext} \\
\textit{zThreads} : \mathbb{F}\ \textit{Thread} \\
\textit{zInterrupts} : \mathbb{F}_1\ \textit{Interrupt} \\
\textit{zExecutionContextToCpu} : \textit{ExecutionContext} ⤀ \textit{CPUID} \\
\textit{zThreadToExecutionContext} : \textit{Thread} ⤔ \textit{ExecutionContext} \\
\textit{zInterruptToExecutionContext} : \textit{Interrupt} ⤔ \textit{ExecutionContext} \\
\hline
\operatorname{ran} \textit{zCpuIdToCpuController} = \textit{cpuControllers} \\
\forall \textit{cpu} : \textit{CPUID} \bullet (\textit{zCpuIdToCpuController}(\textit{cpu})).\textit{cpuId} = \textit{cpu} \\
\bigcap\{\textit{cpuC} : \textit{cpuControllers} \bullet \textit{cpuC}.\textit{threads}\} = \emptyset \\
\bigcup\{\textit{cpuC} : \textit{cpuControllers} \bullet \textit{cpuC}.\textit{threads}\} = \textit{zThreads} \\
\bigcap\{\textit{cpuC} : \textit{cpuControllers} \bullet \textit{cpuC}.\textit{interrupts}\} = \emptyset \\
\bigcup\{\textit{cpuC} : \textit{cpuControllers} \bullet \textit{cpuC}.\textit{interrupts}\} = \textit{zInterrupts} \\
\textit{zExecutionContexts} = \\
\qquad \{t : \textit{zThreads} \bullet t.\textit{executionContext}\} \cup \{i : \textit{zInterrupts} \bullet i.\textit{executionContext}\} \\
\{t : \textit{zThreads} \bullet t.\textit{executionContext}\} \cap \{i : \textit{zInterrupts} \bullet i.\textit{executionContext}\} = \emptyset \\
\operatorname{dom} \textit{zExecutionContextToCpu} = \textit{zExecutionContexts} \\
\operatorname{dom} \textit{zThreadToExecutionContext} = \textit{zThreads} \\
\operatorname{ran} \textit{zThreadToExecutionContext} = \{t : \textit{zThreads} \bullet t.\textit{executionContext}\} \\
\operatorname{dom} \textit{zInterruptToExecutionContext} = \textit{zInterrupts} \\
\operatorname{ran} \textit{zInterruptToExecutionContext} = \{i : \textit{zInterrupts} \bullet i.\textit{executionContext}\} \\
\forall \textit{et} : \textit{zExecutionContexts} \bullet \textit{zExecutionContextToCpu}(\textit{et}) = \textit{et}.\textit{cpuId} \\
\bigcap\{\textit{et} : \textit{zExecutionContexts} \bullet \textit{et}.\textit{executionStack}\} = \emptyset \\
\hline
\end{array}
$$

An execution context can correspond to a software-scheduled thread or to an interrupt. Interrupts are seen as “hardware-scheduled” threads that have higher priority than software-scheduled threads. Thus, an interrupt can be seen as a hardware thread that can preempt the highest priority software thread.

Invariants:

- Every thread must be assigned to a CPU. This is done at thread creation time and the thread cannot be moved to another CPU.

- Every interrupt source must be assigned to a CPU and must always be served by that CPU.
- The same thread cannot be assigned to more than one CPU.
- The same interrupt source cannot be assigned to more than one CPU.
- Every execution context is assigned to a fixed CPU. The same execution context cannot be assigned to two different CPUs. Every CPU has at least one execution context (if nothing else, its idle thread).
- Synchronization between threads and interrupts via codvars is only allowed for threads and interrupts running on the same CPU. That is, a thread cannot be signaled from an ISR running on a different CPU. In other words, a thread cannot wait for an interrupt assigned to a different CPU.

2.5.2 CPU Controllers

$$
\begin{array}{|l}
\textit{CpuController} \\
\hline
\textit{ThreadScheduler} \\
\textit{cpuId} : \textit{CPUID} \\
\textit{zExecutionContexts} : \mathbb{F}_1\ \textit{ExecutionContext} \\
\textit{preemptedBy} : \textit{ExecutionContext} \mathrel{\rightarrowtail\!\!\!\!\!\!\!+} \textit{ExecutionContext} \\
\textit{timers} : \mathbb{F}\ \textit{Timer} \\
\textit{zIterruptChannelToInterrupt} : \textit{INTERRUPT\_CHANNEL} \rightarrowtail \textit{Interrupt} \\
\textit{interrupts} : \mathbb{F}_1\ \textit{Interrupt} \\
\textit{tickTimerInterrupt} : \textit{Interrupt} \\
\textit{runningExecutionContext} : \textit{ExecutionContext} \\
\textit{nestedInterruptCount} : 0 \mathrel{..} \textit{kMaxNumInterruptChannelsPerCpu} \\
\textit{activeInterruptsBitMap} : \mathbb{F}\ \textit{INTERRUPT\_CHANNEL} \\
\textit{activeInterrupts} : \mathbb{F}\ \textit{Interrupt} \\
\hline
\operatorname{ran} \textit{zIterruptChannelToInterrupt} = \textit{interrupts} \\[1ex]
\textit{zExecutionContexts} = \\
\qquad \{ t : \textit{threads} \bullet t.\textit{executionContext} \} \cup \{ i : \textit{interrupts} \bullet i.\textit{executionContext} \} \\[1ex]
\{ t : \textit{threads} \bullet t.\textit{executionContext} \} \cap \{ i : \textit{interrupts} \bullet i.\textit{executionContext} \} = \emptyset \\[1ex]
\forall\, et : \textit{zExecutionContexts} \bullet et.\textit{cpuId} = \textit{cpuId} \\[1ex]
\textit{activeInterrupts} = \{ iv : \textit{activeInterruptsBitMap} \bullet \textit{zIterruptChannelToInterrupt}(iv) \} \\[1ex]
\textit{nestedInterruptCount} = \#\textit{activeInterrupts} \\[1ex]
\textit{nestedInterruptCount} = 0 \Leftrightarrow \\
\qquad \textit{runningExecutionContext} \in \{ t : \textit{zRunnableThreads} \bullet t.\textit{executionContext} \} \\[1ex]
\textit{nestedInterruptCount} > 0 \Leftrightarrow \\
\qquad \textit{runningExecutionContext} \in \{ i : \textit{activeInterrupts} \bullet i.\textit{executionContext} \} \\
\hline
\end{array}
$$

Invariants:

- There can be more than one interrupt with the same interrupt priority. Interrupt scheduling is done by hardware. For the LPC2478, this is done by the Vectored Interrupt Controller (VIC). For ARMv7-A and ARMv7-R processors, the ARM generic interrupt controller (GIC) [4] is used. For ARMv7-M processors, the Nested Vectored Interrupt controller (NVIC) is used.
- The same interrupt cannot be nested.

ThreadScheduler represents the state variables of the Per-CPU thread scheduler.

ThreadScheduler

$$
\begin{aligned}
& zThreadIdToThread : THREAD\_ID \rightarrowtail\!\!\!\!\!\!+\ Thread \\
& threads : \mathbb{F}_1\ Thread \\
& zUserThreads : \mathbb{F}\ Thread \\
& zSystemThreads : \mathbb{F}_1\ Thread \\
& idleThread : Thread \\
& runningThread : Thread \\
& runnableThreadPrioritiesBitMap : \mathbb{F}_1\ THREAD\_PRIO \\
& runnableThreadQueues : THREAD\_PRIO \rightarrowtail ThreadQueue \\
& zRunnableThreads : \mathbb{F}_1\ Thread
\end{aligned}
$$

$$
\begin{aligned}
& \operatorname{ran} zThreadIdToThread = threads \\[1ex]
& zRunnableThreads = \\
& \qquad \bigcup\{i : THREAD\_PRIO \bullet \operatorname{ran}(runnableThreadQueues(i)).zElements\} \\[1ex]
& zRunnableThreads \neq \emptyset \land zRunnableThreads \subseteq threads \\[1ex]
& threads = zUserThreads \cup zSystemThreads \\[1ex]
& zUserThreads \cap zSystemThreads = \emptyset \\[1ex]
& \forall t : zSystemThreads \bullet t.executionContext.cpuMode = kPrivilegedThreadMode \\[1ex]
& \forall t : zUserThreads \bullet \\
& \qquad t.executionContext.cpuMode \in \\
& \qquad\qquad \{kUnprivilegedThreadMode, kPrivilegedThreadMode\} \\[1ex]
& idleThread \in zSystemThreads \\[1ex]
& zThreadIdToThread(0) = idleThread \\[1ex]
& runningThread \in zRunnableThreads \land runningThread.state = kRunning \\[1ex]
& \forall t : zRunnableThreads \setminus \{runningThread\} \bullet t.state = kRunnable \\[1ex]
& \forall t : threads \setminus zRunnableThreads \bullet \\
& \qquad t.state \notin \{kRunnable, kRunning\} \\[1ex]
& \operatorname{ran}(runnableThreadQueues(kLowestThreadPriority)).zElements = \{idleThread\} \\[1ex]
& \forall t : threads \bullet \\
& \qquad runningThread.currentPriority \geq t.currentPriority \\[1ex]
& \forall prio : runnableThreadPrioritiesBitMap \bullet prio \in \operatorname{dom} runnableThreadQueues
\end{aligned}
$$

Invariants:

- The running thread is always the highest priority thread. There can be more than one thread with the same thread priority. Threads of equal priority are time-sliced in a round-robin fashion.

- Each CPU has an idle thread. The idle thread has the lowest priority and cannot get blocked on any mutex or condvar, but it is the only thread that can execute an instruction that stops the processor until an interrupt happens.

$\mathit{ThreadQueue}$

$\mathit{GenericLinkedList}[\mathit{Thread}]$

2.5.3 ExecutionContext

$$
\begin{array}{|l}
\hline
\textit{ExecutionContext} \\
\textit{cpuRegisters} : \mathbb{F}_1\ \textit{WORD\_LOCATION} \\
\textit{stackPointer} : \textit{WORD\_LOCATION} \\
\textit{cpuId} : \textit{CPUID} \\
\textit{cpuMode} : \textit{CPU\_MODE} \\
\textit{contextType} : \textit{EXECUTION\_CONTEXT\_TYPE} \\
\textit{executionStack} : \textit{ADDRESS} \rightarrowtail\!\!\!\!\!\mapsto \textit{WORD\_LOCATION} \\
\textit{exeStackTopEnd} : \textit{ADDRESS} \\
\textit{exeStackBottomEnd} : \textit{ADDRESS} \\
\textit{contextName} : \textit{STRING} \\
\hline
\textit{stackPointer} \in \textit{cpuRegisters} \\
\textit{kWordValue}(\textit{stackPointer}) \in \operatorname{dom} \textit{executionStack} \\
\textit{exeStackTopEnd} < \textit{exeStackBottomEnd} \\
\textit{exeStackTopEnd} \mathrel{..} \textit{exeStackBottomEnd} \subset \textit{kValidRamWordAddresses} \\
\operatorname{dom} \textit{executionStack} = \textit{exeStackTopEnd} + 1 \mathrel{..} \textit{exeStackBottomEnd} \\
\operatorname{dom} \textit{executionStack} \subset \textit{kValidRamWordAddresses} \\
\operatorname{dom} \textit{executionStack} \cap \textit{kReadOnlyAddresses} = \emptyset \\
\hline
\end{array}
$$

2.5.4 Threads

$$
\begin{array}{|l}
\hline
\textit{Thread} \\
\textit{executionContext} : \textit{ExecutionContext} \\
\textit{threadID} : \textit{THREAD\_ID} \\
\textit{threadFunction} : \textit{kExecutableAddresses} \\
\textit{state} : \textit{THREAD\_STATE} \\
\textit{basePriority} : \textit{THREAD\_PRIO} \\
\textit{currentPriority} : \textit{THREAD\_PRIO} \\
\textit{listNode} : \textit{LIST\_NODE} \\
\textit{deadlineToRun} : \mathbb{N} \\
\hline
\textit{currentPriority} \geq \textit{basePriority} \\
\textit{executionContext.contextType} = \textit{kThreadContext} \\
\textit{executionContext.cpuMode} \in \{\textit{kUnprivilegedThreadMode}, \textit{kPrivilegedThreadMode}\} \\
\#\textit{executionContext.executionStack} = \textit{kThreadStackSizeInWords} \\
\hline
\end{array}
$$

User-created threads will run in the ARM user mode and system internal threads will run in the ARM system mode. This is to prevent user threads to execute privileged instructions. When user threads call the system APIs, API wrappers that execute the ARM system call instruction are actually invoked. Then, the SWI exception handler, which gets invoked in supervisor mode, changes the CPU mode to system mode and call the corresponding kernel service. When the kernel service returns to the SWI exception handler, it switches back to supervisor mode and returns form the exception, causing the CPU mode to go back to user mode again.

Invariants:

- The current priority of a thread can never be lower than its base priority. The current priority can be higher than the base priority, due to priority inheritance, when the thread owns a mutex a higher priority thread is waiting for.

2.5.5 Interrupts

Interrupt

$executionContext : ExecutionContext$
$interruptChannel : INTERRUPT\_CHANNEL$
$isrFunction : kExecutableAddresses$

$executionContext.contextType = kInterruptContext$

$executionContext.cpuMode = kInterruptMode$

$\#executionContext.executionStack = kInterruptStackSizeInWords$

Interrupt execution contexts run in the ARM IRQ mode. To ensure that a higher priority interrupt is not delayed by a lower priority interrupt, nested interrupts will be supported. After clearing the interrupt source, all interrupt service routines (ISRs), re-enable IRQ interrupts in the ARM core, but continue running in IRQ mode. Note that the same interrupt cannot be raised again until we finish servicing the current one, as the interrupt controller is expected to only raise interrupts with higher priority than the current one being served. (The last step in servicing an interrupt is to notify the interrupt controller of the completion of servicing the interrupt).

2.5.6 Timers

Timer

$counter : \mathbb{N}$

2.5.7 Mutexes

Mutex

$waitingThreads : ThreadQueue$
$synchronizationScope : SYNCHRONIZATION\_SCOPE$

$synchronizationScope \in \{kLocalCpuThreadOnly, kInterCpuThreadOnly\}$

When a mutex is released and another thread is waiting to acquire it, the ownership of the mutex is transferred to the first waiter, and this waiter is made runnable. This is so that if the previous owner has higher priority and tries to acquire it again, it will get blocked. Otherwise, the highest priority thread will keep running, acquiring and releasing the mutex without giving a chance to the low priority waiting thread to ever get it.

The queue of waiters on a mutex is strictly FIFO, not priority based. This is to ensure fairness for lower priority threads. Otherwise, lower priority threads may starve waiting to get the mutex, as higher priority threads keep acquiring it first.

The primitive to acquire a mutex returns with error if the reliability monitor detects a deadlock, to prevent the caller from waiting indefinitely.

2.5.8 Condition Variables

Condvar

$waitingThreads : ThreadQueue$
$synchronizationScope : SYNCHRONIZATION\_SCOPE$

Besides the traditional condvar "wait" primitive, there is an additional "Wait for interrupt" primitive, intended to be used to synchronize a waiting thread with an ISR that is supposed to signal the corresponding condvar on which the thread is waiting. The waiting thread must have interrupts disabled in the ARM processor, when it calls "wait for interrupt".

If more than one thread is waiting on the condvar, the "signal" primitive will wake up the first thread in the condvar's queue. The "broadcast" primitive wakes up all the waiting threads.

There is a variation of the "wait" primitive that includes a timeout.

McRTOS will not provide semaphore primitives as part of its user APIs, as semaphores can be easily implemented using condition variables and mutexes, for semaphores used only by threads. For semaphores signaled from ISRs, they can be implemented with a combination of condition variables and disabling interrupts, since mutexes cannot be used in ISRs. In this case, the thread waiting on the condition variable to be signaled by an ISR, disables interrupts before checking the condition and calls the "wait for interrupt" primitive, if the condition has not been met. Otherwise, missed "wake-ups" could happen due to a race condition between the thread and the ISR.

2.5.9 Message Channels

MessageChannel

$$GenericCircularBuffer[WORD\_LOCATION]$$

2.5.10 Object Pools

ObjectPool

$$\begin{array}{l}
objects : \mathbb{F}_1\ ADDRESS \\
zNumObjects : \mathbb{N}_1 \\
freeObjects : FreeList \\
zNumFreeObjects : \mathbb{N} \\
objectType : OBJECT\_TYPE \\
\hline
\#objects = zNumObjects \\
freeObjects.numEntries = zNumObjects \\
freeObjects.entriesFilled = zNumFreeObjects
\end{array}$$

FreeList

$$GenericCircularBuffer[ADDRESS]$$

2.5.11 Reliability Monitor

ReliabilityMonitor

$\mathit{failureTraceLog} : \mathit{FailureTraceBuffer}$

FailureTraceBuffer

$\mathit{GenericCircularBuffer}[\mathit{Failure}]$

Failure

$\mathit{location} : \mathit{ADDRESS}$
$\mathit{timeStamp} : \mathbb{N}$
$\mathit{stackTrace} : \mathrm{iseq}\ \mathit{WORD\_LOCATION}$

Failures to be detected by the Reliability Monitor:

- Interrupts have been disabled for too long (longer than a predefined threshold).
- Mutex deadlock (initial implementation may simply use a timer-based detection approach to detect if a mutex has been held for too long)
- Thread starvation: a runnable thread other than the idle thread has not been able to get the CPU for too long.
- A time-critical thread has missed its next deadline to run.
- Failure detectors will be placed through the code of McRTOS in the form of production-code “asserts” (as opposed to debug-only asserts that are compiled-out from production code).
- Regular error handling in the code will call the Reliability monitor to capture failure data associated with the error. The reliability monitor will assign a unique error to each error detected by error handling logic. The purpose of these error codes is to uniquely identify the point in the source code where the error originated.

2.5.12 Generic Data Structures

2.5.12.1 Generic Linked Lists

$GenericLinkedList[ElementType]$

$listAnchor : LIST\_NODE$
$numNodes : \mathbb{N}$
$zNodes : \mathbb{F}\, LIST\_NODE$
$zElements : \mathrm{iseq}\, ElementType$
$zNodeToElem : LIST\_NODE \rightarrowtail\!\!\!\!\!\!\!\twoheadrightarrow ElementType$
$zNextNode : LIST\_NODE \rightarrowtail\!\!\!\!\!\!\!\twoheadrightarrow LIST\_NODE$
$zPrevNode : LIST\_NODE \rightarrowtail\!\!\!\!\!\!\!\twoheadrightarrow LIST\_NODE$
$zNodeToListAnchor : LIST\_NODE \rightarrowtail\!\!\!\!\!\!\!\twoheadrightarrow LIST\_NODE$

$listAnchor \notin zNodes$

$numNodes = \# zNodes$

$\mathrm{dom}\, zNodeToElem = zNodes$

$\mathrm{ran}\, zNodeToElem = \mathrm{ran}\, zElements$

$\mathrm{dom}\, zNextNode = zNodes \cup \{listAnchor\}$

$\mathrm{ran}\, zNextNode = zNodes \cup \{listAnchor\}$

$\mathrm{dom}\, zPrevNode = \mathrm{dom}\, zNextNode$

$\mathrm{ran}\, zPrevNode = \mathrm{ran}\, zNextNode$

$\# zElements = \# zNodes$

$head\ zElements = zNodeToElem(zNextNode(listAnchor)) \Leftrightarrow zElements \neq \emptyset$

$last\ zElements = zNodeToElem(zPrevNode(listAnchor)) \Leftrightarrow zElements \neq \emptyset$

$head\ zElements = last\ zElements \Leftrightarrow \# zElements = 1$

$\forall x : zNodes \bullet$
$\quad zPrevNode(zNextNode(x)) = x \wedge zNextNode(zPrevNode(x)) = x \wedge$
$\quad zNodeToListAnchor(x) = listAnchor$

$\forall x : zNodes \bullet$
$\quad zNextNode^{\# zNodes + 1}(x) = x \wedge zPrevNode^{\# zNodes + 1}(x) = x$

$\forall x : zNodes;\ k : 1 \mathinner{..} \# zNodes \bullet$
$\quad zNextNode^{k}(x) \neq x \wedge zPrevNode^{k}(x) \neq x$

$zNextNode(listAnchor) = zNodeToElem^{\sim}(zElements(0))$

$zPrevNode(listAnchor) = zNodeToElem^{\sim}(last(zElements))$

$zNextNode(listAnchor) = listAnchor \Leftrightarrow zNodes = \emptyset$

$zPrevNode(listAnchor) = listAnchor \Leftrightarrow zNextNode(listAnchor) = listAnchor$

$zNextNode(listAnchor) = zPrevNode(listAnchor) \Leftrightarrow \# zNodes \leq 1$

2.5.12.2 Generic Circular Buffers

$$
\begin{array}{|l}
\hline
\mathit{GenericCircularBuffer}[\mathit{EntryType}] \\
\mathit{zEntries} : \mathrm{iseq}\, \mathit{EntryType} \\
\mathit{numEntries} : \mathbb{N}_1 \\
\mathit{entriesFilled} : \mathbb{N} \\
\mathit{readCursor} : \mathbb{N} \\
\mathit{writeCursor} : \mathbb{N} \\
\mathit{synchronizationScope} : \mathit{SYNCHRONIZATION\_SCOPE} \\
\mathit{mutex} : \mathit{Mutex} \\
\mathit{notEmptyCondvar} : \mathit{Condvar} \\
\mathit{notFullCondvar} : \mathit{Condvar} \\
\hline
\#\mathit{zEntries} = \mathit{numEntries} \\
\mathit{entriesFilled} \in 0 \mathrel{..} \mathit{numEntries} \\
\mathit{readCursor} \in 0 \mathrel{..} \mathit{numEntries} - 1 \\
\mathit{writeCursor} \in 0 \mathrel{..} \mathit{numEntries} - 1 \\
\mathit{writeCursor} = \mathit{readCursor} \Leftrightarrow \\
\qquad (\mathit{entriesFilled} = 0 \lor \mathit{entriesFilled} = \mathit{numEntries}) \\
\mathit{notEmptyCondvar} \neq \mathit{notFullCondvar} \\
\mathit{notEmptyCondvar}.\mathit{synchronizationScope} = \mathit{synchronizationScope} \\
\mathit{notFullCondvar}.\mathit{synchronizationScope} = \mathit{synchronizationScope} \\
\hline
\end{array}
$$

If *synchronizationScope* is *kLocalCpuInterruptAndThread*, the circular buffer operations disable interrupts instead of using the circular buffer's mutex. If a circular buffer is empty, a reader will block until the buffer is not empty. Three behaviors are possible for writers when a circular buffer is full: block until there is room to complete the write, drop the item to be written, overwrite the oldest entry with the new item.

Chapter 3

Code Quality Guidelines

3.1 Compliance with Industry Standards for Code Reliability

All the C code of McRTOS will follow the MISRA C 2012 standard [8] and the CERT C Secure Coding Standard [9].

3.2 Doxygen Code Documentation

All functions, types, constants and global variables will have a preceding comment block in Doxygen format [10], so that API documentation can be generated automatically from code comments.

3.3 Opaque Objects return by User-level APIs

All the “create” APIs return opaque pointers (pointer to incomplete structs). They point to objects allocated from internal object pools statically allocated (`malloc()` is not used at all). Entries in these object pools are cache-line aligned, to avoid cache line trashing when threads running on different processors keep modifying neighboring entries. The returned pointers are“scrambled” so that the user cannot dereference them directly. A simple approach to scramble a pointer is to rotate its bits left or right 2 or 3 bits, using an inline assembly instruction. When an API needs to dereference a pointer, it will “un-scramble” it first, by doing the opposite rotation.

3.4 Design-by-Contract Principles

Design-by-Contract principles [11] will be followed to systematically identify places in the code where to put “asserts”. Assertion checks will stay in production code, as opposed to being compiled out from production builds.

3.5 Test-Driven Development

Test-driven development practices [12] will be used to develop the code of McRTOS. In particular the CppUTest [13] unit-testing framework will be used to create automated unit tests.

Bibliography

[1] Steve Furber, "ARM System-on-Chip Architecture", second edition, Addison-Wesley, 2000
`http://www.amazon.com/ARM-System-Chip-Architecture-Edition/dp/0201675196`

[2] Andrew Sloss et al, "ARM System Developer's Guide: Designing and Optimizing System Software", Morgan Kaufmann, 2004
`http://www.amazon.com/ARM-System-Developers-Guide-Architecture/dp/1558608745`

[3] ARM, "Cortex-A Series Programmers Guide", version 3.0, ARM, 2012
`http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.den0013-/index.html`

[4] ARM, "ARM Generic Interrupt Controller Architecture Specification Architecture", version 2.0, ARM, 2011
`http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.ihi0048b/index.html`

[5] Mike Spivey, "The Z Reference Manual", second edition, Prentice-Hall, 1992
`http://spivey.oriel.ox.ac.uk/~mike/zrm/zrm.pdf`

[6] Jonathan Jacky, "The Way of Z", Cambridge Press, 1997
`http://staff.washington.edu/jon/z-book/index.html`

[7] Mike Spivey, "The Fuzz checker"
`http://spivey.oriel.ox.ac.uk/mike/fuzz`

[8] MISRA, "MISRA C:2012"
`http://www.misra.org.uk/LinkClick.aspx?fileticket=SAG1iH3YwNE%3d&tabid=59`

[9] Software Engineering Institute, "CERT C Secure Coding Standard"
`https://www.securecoding.cert.org/confluence/display/seccode/CERT+C+Secure+Coding+Standard`

[10] Dimitri van Heesch, “Doxygen Documentation Generator”
`http://doxygen.org`

[11] Bertrand Meyer, “Touch of Class: Learning to Program Well with Objects and Contracts”, Springer, 2009
`http://www.amazon.com/dp/3540921443`

[12] James W. Grenning, “Test Driven Development for Embedded C”, Pragmatic Bookshelf, 2011
`http://www.amazon.com/Driven-Development-Embedded-Pragmatic-Programmers/dp/193435662X`

[13] CppUTest Project, “CppUTest unit testing and mocking framework for C/C++”
`http://cpputest.org`