Design of the *HiRTOS* Multi-core Real-Time Operating System

Germán Rivera
jgrivera67@gmail.com

June 8, 2024

Contents

Chapter 1

Introduction

This document describes the design of *HiRTOS* (*"High Integrity"* RTOS), a real-time operating system kernel (RTOS) written in SPARK Ada. HiRTOS targets safety-critical and security-sensitive embedded software applications that run in small multi-core microcontrollers. HiRTOS was designed using the Z notation, as a methodical way to capture correctness properties that can be expressed as programming contracts in SPARK Ada. Z is a software modeling notation based on discrete mathematics structures (such as sets, relations and functions) and predicate logic.

Although there are several popular RTOSes for embedded applications that run on small multi-core microcontrollers, most of them are not designed with high-integrity applications in mind, and are typically written in C, a notoriously unsafe language. So, it would be desirable to have an RTOS specifically designed for high-integrity applications, and written in a safer language, like Ada or its subset SPARK Ada, even if application code is written in C/C++. Modern versions of the Ada and SPARK languages have programming-by-contract constructs built-in in the language, which allows the programmer to express correctness properties, such as preconditions, postconditions, object state invariants and loop invariants, as code. However, one challenge when doing programming-by-contract is to be aware of all the correctness conditions that can be asserted in programming contracts. Describing software design in a formal notation, such as the Z notation [1, 2, 3], can help identify/elicit such correctness conditions in a more thorough and methodical way than just writing code.

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 correctness properties, in a systematic way. The *HiRTOS* Z model described here was checked with the `fuzz` tool [4], a Z type-checker, that catches Z type mismatches in predicates.

The code of *HiRTOS* is written in SPARK Ada [5], a high integrity subset of

the Ada programming language. The code of HiRTOS is available in GitHub at `https://github.com/jgrivera67/HiRTOS`. HiRTOS data types were modeled in Z at a level of abstraction that can be mapped directly to corresponding data types in SPARK Ada. This guided design decisions at the code level, resulting in a completely pointer-less implementation, that leverages the various Ada mechanisms to avoid having to use pointers.

Also, the HiRTOS thread scheduler was modeled in TLA+/PlusCal [15, 16] and the hiRTOS TLA+ model was verified using the TLC model checker [17].

Chapter 2

HiRTOS Overview

2.1   RTOS Major Design Decisions

- For API simplicity, mutexes and condition variables [7, 8] are the only synchronization primitives in *HiRTOS*, similar to the thread synchronization primitives of the C11 standard library [9]. Other synchronization primitives such as semaphores, event flags and message queues can be implemented on top of mutexes and condition variables.

- Unlike C11 mutexes, *HiRTOS* mutexes can change the priority of the thread owning the mutex. *HiRTOS* mutexes support both priority inheritance and priority ceiling [10].

- Unlike C11 condition variables, *HiRTOS* condition variables can also be waited on while having interrupts disabled, not just while holding a mutex. This prevents missing "thread wakeups", when signaling condition variables from interrupt service routines (ISRs), making it unnecessary to use semaphores for this purpose.

- *HiRTOS* atomic levels can be used to disable the thread scheduler or to disable interrupts at and below a given priority or to disable all interrupts.

- In a multi-core platform, there is one *HiRTOS* instance per CPU Core. Each *HiRTOS* instance is independent of each other. No resources are shared between *HiRTOS* instances. No communication/synchronization between CPU cores is supported by the current version of *HiRTOS*. If an application needs inter-core communication, that would need to be provided outside of *HiRTOS*, using doorbell interrupts and mailboxes or shared memory, for example.

- Threads are bound to the CPU core in which they were created, for the lifetime of the thread. That is, no thread migration between CPU cores is supported.

- All RTOS objects such as threads, mutexes and condition variables are allocated internally by *HiRTOS* from statically allocated internal object pools. These object pools are just RTOS-private global arrays of the corresponding RTOS object types, sized at compile time via configuration parameters, whose values are application-specific. RTOS object handles provided to application code are just indices into these internal object arrays. No actual RTOS object pointers exposed to application code. No dynamic allocation/deallocation of RTOS objects is supported and no static allocation of RTOS objects in memory owned by application code is supported either. All this prevents HiRTOS internal state from being accidentally or intentionally corrupted from application code. RTOS objects can only be manipulated through the HiRTOS APIs.

- All application threads run in unprivileged mode by default. In unprivileged mode, a thread can only access its own stack. To access global variables or MMIO space, application threads must explicitly request either read-only or read-write permission to *HiRTOS* via a system call.

- ISRs are seen as hardware-scheduled threads that have higher priority than all software-scheduled threads. They can only be preempted by higher-priority ISRs. They cannot block waiting on mutexes or condition variables.

2.2 Separation Kernel Major Design Decisions

Besides being a fully functional RTOS, HiRTOS can be used as a separation kernel [11, 12]. A separation kernel can be seen as an RTOS that schedules partitions instead of threads. A partition is a spatial and temporal separation/isolation unit on which a bare-metal or RTOS-based firmware binary runs. A separation kernel requires that the target CPU supports hypervisor privilege mode and two-stage memory protection.

- In a multi-core platform, there is one separation kernel instance per CPU Core. Each instance is independent of each other. No resources are shared between separation kernel instances. No communication between CPU cores is supported yet. If needed by the application, inter-core communication would need to be provided outside of the HiRTOS separation kernel, using doorbell interrupts and mailboxes or shared memory, for example.

- Each separation-kernel instance consists of one or more partitions. A partition is a spatial and temporal separation/isolation unit on which a bare-metal or RTOS-based firmware binary runs. Each partition consists of one or more disjoint address ranges covering portions of RAM and MMIO space that only that partition can access. Isolation is enforced by a hypervisor-level memory protection unit (MPU). Also, each partition has its own interrupt vector table and its own set of physical interrupts. So, physical peripherals can be assigned to individual partitions (discrete device assignment). However, no device or

memory virtualisation is supported, as the target machine for the HiRTOS separation kernel is not required to have neither an MMU nor an IOMMU. The CPU core is time-sliced among the partitions running on the same separation kernel instance.

- Partitions are bound to the CPU core in which they were created. That is, no partition migration between CPU cores is supported.

- Partitions are created at boot time before starting the partition scheduler on the corresponding CPU core. Partitions cannot be destroyed or terminated.

- The separation kernel code itself runs in hypervisor privilege mode. All partitions run at a privilege lower than hypervisor mode. Partitions can communicate with the separation kernel via hypervisor calls and via traps to hypervisor mode triggered from special machine instructions such as *WFI*. The separation kernel can communicate with partitions, by forwarding interrupts targeted to the corresponding partition.

2.3   HiRTOS Code Architecture

To have wider adoption of an RTOS written in bare-metal Ada, providing a C/C++ programming interface is a must. Indeed, multiple interfaces or "skins" can be provided to mimic widely popular RTOSes such as FreeRTOS [13] and RTOS interfaces such as the CMSIS RTOS2 API [14]. As shown on figure 2.1, HiRTOS has a C/C++ interface layer that provides a FreeRTOS skin and and a CMSIS RTOS2 skin. Both skins are implemented on top of a native C skin. The native C skin is just a thin C wrapper that consists of a C header file containing the C functions prototypes of the corresponding Ada subprograms of the SPARK Ada native interface of HiRTOS.

In addition to the Ada native and C/C++ interfaces, HiRTOS could also provide an Ada runtime library (RTS) skin, as shown on figure 2.2, so that baremetal Ada applications that use Ada tasking features can run on top of HiRTOS. This can be especially useful, given the limited number of microcontroller platforms for which an available bare-metal Ada runtime library is provided with the GNAT Ada compiler.

HiRTOS has been architected to be easily portable to any single-core or multi-core microcontroller or bare metal platform for which a GNAT Ada cross compiler is available. All platform-dependent code is isolated in the HiRTOS porting layer, which provides platform-independent interfaces to the rest of the HiRTOS code. To avoid any dependency on a platform-specific bare-metal Ada runtime library, provided by the compiler, HiRTOS sits on top of a platform-independent portable minimal Ada runtime library. Currently, two porting layers have been implemented: one for the ARM Cortex-R52 multi-core processor, and the other for the RISCV-based ESP32-C3 microcontroller. To port HiRTOS to a new target platform, the platform-specific components of the porting layer would need to be implemented for the new platform.
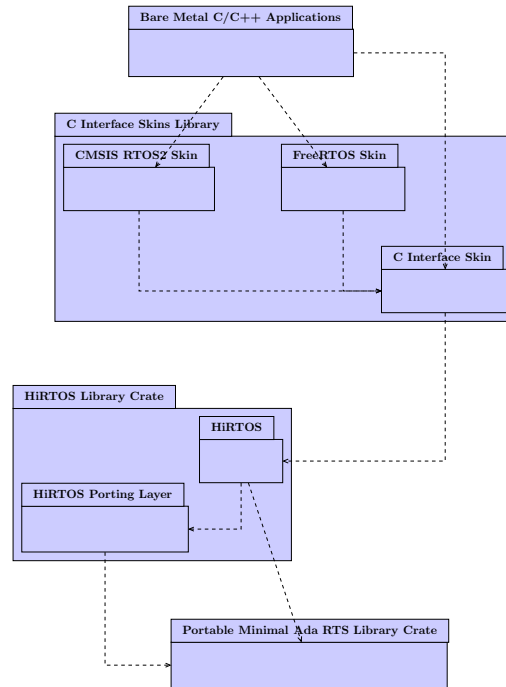
Figure 2.1: HiRTOS Code Architecture for C/C++ Applications
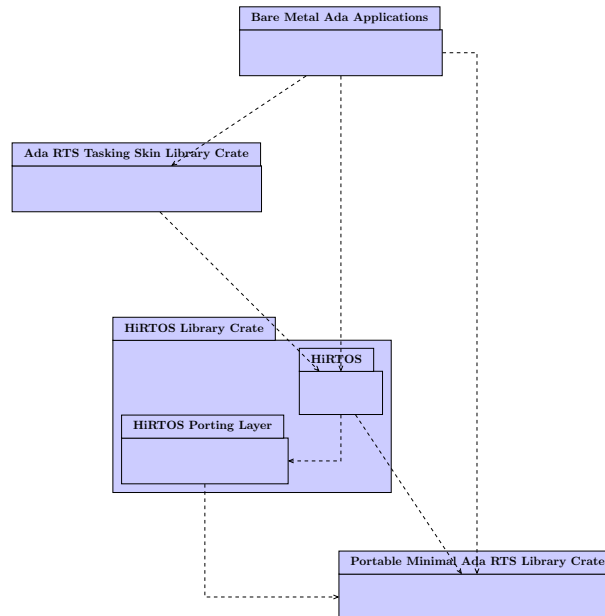
Figure 2.2: HiRTOS Code Architecture for Ada Applications

Figure 2.3 shows the major code components of HiRTOS. The HiRTOS code base is structured in three conceptual layers. The *HiRTOS API* layer, the *HiRTOS internals layer* and the *HiRTOS porting layer*.

The *HiRTOS API* layer contains the HiRTOS public interface components. The `HiRTOS_Interrupt_Handling` Ada package contains the services to be invoked from top-level interrupt handlers to notify HiRTOS of entering an exiting interrupt context. `HiRTOS_Memory_Protection` contains the services to protect ranges of memory and MMIO space. `HiRTOS_Thread` contains the services to create and manage threads.

The *HiRTOS internals layer* contains HiRTOS-private components that are hardware-independent.

The *HiRTOS porting layer* contains hardware-dependent components that provide hardware-independent interfaces to upper HiRTOS layers.
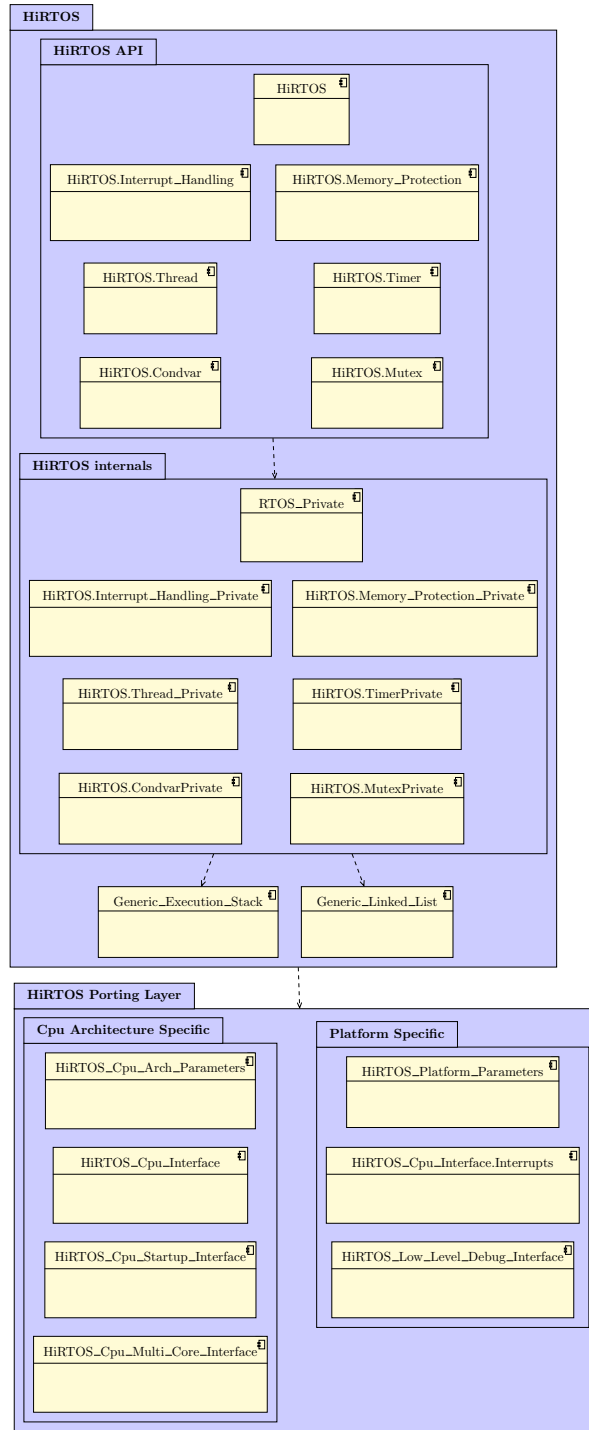
Figure 2.3: HiRTOS Code Components

Chapter 3

HiRTOS Z Specification

3.1 HiRTOS Data Structures

3.1.1 Z Naming Conventions

The following naming conventions are used in the Z model of *HiRTOS*:

- Z Primitive types are in uppercase.

- Z Composite types (schema types) start with uppercase.

- Z constants and variables start with lower case.

- Identifiers that start with the z prefix are meant to be modeling-only entities that do not physically correspond to code-level entities.

3.1.2 *HiRTOS* Configuration Parameters

Constants defined here represent compile-time configuration parameters for *HiRTOS*.

$$
\begin{array}{l}
maxNumThreads : \mathbb{N}_1 \\
maxNumMutexes : \mathbb{N}_1 \\
maxNumCondvars : \mathbb{N}_1 \\
maxNumTimers : \mathbb{N}_1 \\
numThreadPriorities : \mathbb{N}_1 \\
numTimerWheelSpokes : \mathbb{N}_1 \\
\hline
maxNumThreads > 2 \\
\\
maxNumCondvars \geq maxNumThreads \\
\\
maxNumTimers \geq maxNumThreads
\end{array}
$$

The minimum number of threads that can be configured per CPU core is 2, which corresponds to the *HiRTOS* pre-defined threads: the idle thread and the tick timer

9

thread. Each thread has a builtin timer, so the minimum number of timers that can be configured is *maxNumThreads*. Also, each thread has a builtin condition variable, so the minimum number of condition variables that can be configured is *maxNumThreads* as well.

3.1.3 *HiRTOS* Target Platform Parameters

Constants defined here represent compile-time target platform parameters for *HiRTOS*.

$numCpus : \mathbb{N}_1$
$minMemoryAddress : \mathbb{N}$
$maxMemoryAddress : \mathbb{N}_1$
$numInterruptPriorities : \mathbb{N}_1$
$maxNumInterrupts : \mathbb{N}_1$

$minMemoryAddress < maxMemoryAddress$

3.1.4 *HiRTOS* Primitive Types

Below are the primitive types used in HiRTOS:

$[CpuIdType]$
$\#CpuIdType = numCpus + 1$
$[ThreadIdType]$
$\#ThreadIdType = maxNumThreads + 1$
$[MutexIdType]$
$\#MutexIdType = maxNumMutexes + 1$
$[CondvarIdType]$
$\#CondvarIdType = maxNumCondvars + 1$
$[TimerIdType]$
$\#TimerIdType = maxNumTimers + 1$
$[InterruptIdType]$
$\#InterruptIdType = maxNumInterrupts + 1$

$invalidCpuId : CpuIdType$
$invalidThreadId : ThreadIdType$
$invalidMutexId : MutexIdType$
$invalidCondvarId : CondvarIdType$
$invalidTimerId : TimerIdType$
$invalidInterruptId : InterruptIdType$

$[CpuRegistersType]$
$ValidCpuIdType == CpuIdType \setminus \{invalidCpuId\}$
$MemoryAddressType ==$
$\quad minMemoryAddress \mathinner{\ldotp\ldotp} maxMemoryAddress$
$nullAddress == 0$
$ValidThreadIdType ==$
$\quad ThreadIdType \setminus \{invalidThreadId\}$
$ValidMutexIdType == MutexIdType \setminus \{invalidMutexId\}$
$ValidCondvarIdType ==$
$\quad CondvarIdType \setminus \{invalidCondvarId\}$
$ValidTimerIdType == TimerIdType \setminus \{invalidTimerId\}$
$ValidInterruptIdType ==$
$\quad InterruptIdType \setminus \{invalidInterruptId\}$
$ThreadPriorityType == 0 \mathinner{\ldotp\ldotp} numThreadPriorities$
$invalidThreadPriority == numThreadPriorities$
$ValidThreadPriorityType ==$
$\quad ThreadPriorityType \setminus \{invalidThreadPriority\}$
$InterruptPriorityType == 0 \mathinner{\ldotp\ldotp} numInterruptPriorities$
$invalidInterruptPriority == numInterruptPriorities$
$ValidInterruptPriorityType ==$
$\quad InterruptPriorityType \setminus \{invalidInterruptPriority\}$
$AtomicLevelType == 0 \mathinner{\ldotp\ldotp} numInterruptPriorities + 1$
$atomicLevelNoInterrupts == min\ AtomicLevelType$
$atomicLevelSingleThread == max\ AtomicLevelType - 1$
$atomicLevelNone == max\ AtomicLevelType$
$InterruptNestingCounterType ==$
$\quad 0 \mathinner{\ldotp\ldotp} numInterruptPriorities$
$ActiveInterruptNestingCounterType ==$
$\quad InterruptNestingCounterType \setminus \{\, 0 \,\}$
$CpuInterruptMaskingStateType ::=$
$\quad cpuInterruptsEnabled \mid$
$\quad cpuInterruptsDisabled$
$CpuPrivilegeType ::= cpuPrivileged \mid cpuUnprivileged$
$MemoryProtectionStateType ::=$
$\quad memoryProtectionOn \mid memoryProtectionOff$
$CpuExecutionModeType ::=$
$\quad cpuExecutingResetHandler \mid$
$\quad cpuExecutingInterruptHandler \mid$
$\quad cpuExecutingThread$
$ThreadStateType ::= threadNotCreated \mid threadSuspended \mid$
$\qquad\qquad threadRunnable \mid threadRunning \mid$
$\qquad\qquad threadBlockedOnCondvar \mid threadBlockedOnMutex$
$ThreadSchedulerStateType ::=$
$\quad threadSchedulerStopped \mid threadSchedulerRunning$

$$ThreadQueueType == \text{iseq } ValidThreadIdType$$
$$MutexListType == \text{iseq } ValidMutexIdType$$
$$TimerWheelSpokeIndexType ==$$
$$0 \,..\, numTimerWheelSpokes$$
$$invalidTimerWheelSpokeIndex ==$$
$$\max TimerWheelSpokeIndexType$$
$$ValidTimerWheelSpokeIndexType ==$$
$$TimerWheelSpokeIndexType \setminus$$
$$\{\, invalidTimerWheelSpokeIndex \,\}$$
$$TimerKindType ::= periodicTimer \mid oneShotTimer$$
$$TimerStateType ::= timerStopped \mid timerRunning$$

For interrupts, lower priority values represent higher priorities. For threads, lower priority values represent lower priorities.

3.1.5 *HiRTOS* Axiomatic Definitions

$$zAddressRange :$$
$$(MemoryAddressType \times MemoryAddressType) \rightarrowtail$$
$$\mathbb{F}_1 \, MemoryAddressType$$

$$\forall\, x, y : MemoryAddressType \mid$$
$$(x, y) \in \text{dom } zAddressRange \bullet$$
$$x < y \wedge zAddressRange(x, y) = x \,..\, y$$

$$zCpuToISRstackAddressRange :$$
$$ValidCpuIdType \rightarrowtail$$
$$(MemoryAddressType \times MemoryAddressType)$$

$$\bigcap \{\, i : ValidCpuIdType \bullet$$
$$zAddressRange(zCpuToISRstackAddressRange(i)) \,\} = \emptyset$$

$$\forall\, i : \text{dom } zCpuToISRstackAddressRange \bullet$$
$$\#(zAddressRange(zCpuToISRstackAddressRange(i))) \geq 2$$

$$interruptPriorities :$$
$$InterruptIdType \rightarrow InterruptPriorityType$$

3.1.6 *HiRTOS* State Variables

The *HiRtos* schema represents the multi-core RTOS state variables (internal data structures). All HiRTOS objects such as threads, mutexes, condition variables and software timers are statically allocated internally by HiRTOS.

─ *HiRtos* ─────────────────────────────

rtosCpuInstances :
    ValidCpuIdType $\rightarrowtail$ *HiRtosCpuInstanceType*
zAllCreatedThreadInstances : $\mathbb{F}$ *ThreadType*
─────────────────────────────
$\#rtosCpuInstances \geq 1$

$\forall\, i : \mathrm{dom}\; rtosCpuInstances \bullet$
    $(rtosCpuInstances(i)).cpuId = i$

$zAllCreatedThreadInstances =$
    $\bigcup \{\, i : ValidCpuIdType \bullet$
        $\mathrm{ran}(rtosCpuInstances(i)).threads \,\}$

$\bigcap \{\, i : ValidCpuIdType \bullet$
    $\mathrm{ran}(rtosCpuInstances(i)).threads \,\} = \emptyset$

$\bigcap \{\, thread : zAllCreatedThreadInstances \bullet$
    $zAddressRange(thread.stack) \,\} = \emptyset$

$\bigcap \{\, i : ValidCpuIdType \bullet$
    $\mathrm{ran}(rtosCpuInstances(i)).mutexes \,\} = \emptyset$

$\bigcap \{\, i : ValidCpuIdType \bullet$
    $\mathrm{ran}(rtosCpuInstances(i)).condvars \,\} = \emptyset$

$\bigcap \{\, i : ValidCpuIdType \bullet$
    $\mathrm{ran}(rtosCpuInstances(i)).timers \,\} = \emptyset$
─────────────────────────────

Per-CPU HiRTOS Instance

The state variables and internal data structures of each per-CPU *HiRTOS* instance
are described below:

HiRtosCpuInstanceType ──────────────────────────────
$cpuId : CpuIdType$
$currentCpuContext : CpuRegistersType$
$threadSchedulerState : ThreadSchedulerStateType$
$currentAtomicLevel : AtomicLevelType$
$currentCpuExecutionMode : CpuExecutionModeType$
$currentThreadId : ThreadIdType$
$timerTicksSinceBoot : \mathbb{N}$
$idleThreadId : ValidThreadIdType$
$tickTimerThreadId : ValidThreadIdType$
$interruptNestingLevelStack : InterruptNestingLevelStackType$
$threads : ValidThreadIdType \nrightarrow\!\!\!\rightarrow ThreadType$
$mutexes : ValidMutexIdType \nrightarrow\!\!\!\rightarrow MutexType$
$condvars : ValidCondvarIdType \nrightarrow\!\!\!\rightarrow CondvarType$
$timers : ValidTimerIdType \nrightarrow\!\!\!\rightarrow TimerType$
$runnableThreadsQueue : ThreadPriorityQueueType$
$timerWheel : TimerWheelType$
$zCpuInterruptMaskingState : CpuInterruptMaskingStateType$
$zCpuPrivilege : CpuPrivilegeType$
$zMemoryProtectionState : MemoryProtectionStateType$
───
$\{\, idleThreadId, tickTimerThreadId \,\} \subseteq \mathrm{dom}\; threads$

$tickTimerThreadId \neq idleThreadId$

$threadSchedulerState = threadSchedulerRunning \Rightarrow$
$(\forall\, threadId : (\mathrm{dom}\; threads \setminus \{\, currentThreadId \,\}) \bullet$
$\quad (threads(threadId)).currentPriority \leq (threads(currentThreadId)).currentPriority)$

$zCpuInterruptMaskingState = cpuInterruptsEnabled \Leftrightarrow$
$\quad currentAtomicLevel > atomicLevelNoInterrupts$

$zCpuInterruptMaskingState = cpuInterruptsDisabled \Rightarrow zCpuPrivilege = cpuPrivileged$

$currentAtomicLevel < atomicLevelNone \Rightarrow zCpuPrivilege = cpuPrivileged$

$\bigcap \{\, t : \mathrm{ran}\; threads \bullet \{\, t.builtinCondvarId \,\} \,\} = \emptyset$

$\bigcap \{\, t : \mathrm{ran}\; threads \bullet \{\, t.builtinTimerId \,\} \,\} = \emptyset$

$\forall\, t : \mathrm{ran}\; threads \bullet (t.id = (threads^{\sim})(t) \wedge$
$(t.ownedMutexes \neq \emptyset \Rightarrow$
$t.currentPriority = max \{\, m : \mathrm{ran}\; t.ownedMutexes \bullet (mutexes(m)).ceilingPriority \,\}))$

$\forall\, m : \mathrm{ran}\; mutexes \bullet m.id = mutexes^{\sim}(m)$

$\forall\, c : \mathrm{ran}\; condvars \bullet c.id = condvars^{\sim}(c)$

$\forall\, ti : \mathrm{ran}\; timers \bullet ti.id = timers^{\sim}(ti)$

$\forall\, p : ValidThreadPriorityType \bullet$
$\quad \forall\, threadId : \mathrm{ran}(runnableThreadsQueue.threadQueues(p)) \bullet$
$\qquad (threads(threadId)).currentPriority = p$
───

─ *ThreadPriorityQueueType* ────────────────

threadQueues :
    ValidThreadPriorityType $\rightarrowtail$ *ThreadQueueType*
waitingThreadsCount : $\mathbb{N}$

─────────

waitingThreadsCount =
$\#(\bigcup\{ p : ValidThreadPriorityType \bullet threadQueues(p) \})$

─ *InterruptNestingLevelStackType* ────────────────

interruptNestingLevels :
    ActiveInterruptNestingCounterType $\rightarrowtail$
    InterruptNestingLevelType
currentInterruptNestingCounter :
    InterruptNestingCounterType
zCpuId : *ValidCpuIdType*

─────────

$\forall\, x : \mathrm{dom}\, interruptNestingLevels \bullet$
    $(interruptNestingLevels(x)).interruptNestingCounter = x$
    $\wedge$
    $(interruptNestingLevels(x)).savedStackPointer \in$
    $zAddressRange(zCpuToISRstackAddressRange(zCpuId))$

$\mathrm{dom}\, interruptNestingLevels =$
$1 \mathrel{..} currentInterruptNestingCounter$

─ *InterruptNestingLevelType* ────────────────

interruptId : *InterruptIdType*
interruptNestingCounter : *ActiveInterruptNestingCounterType*
savedStackPointer : *MemoryAddressType*
atomicLevel : *AtomicLevelType*

─────────

$atomicLevel \leq interruptPriorities(interruptId)$

─ *TimerWheelType* ────────────────

wheelSpokesHashTable :
    ValidTimerWheelSpokeIndexType $\rightarrow \mathbb{F}\, ValidTimerIdType$
currentWheelSpokeIndex : *ValidTimerWheelSpokeIndexType*

─────────

$\forall\, i,j : ValidTimerWheelSpokeIndexType \mid i \neq j \bullet$
    $wheelSpokesHashTable(i) \cap wheelSpokesHashTable(j) = \emptyset$

TimerType

$id : TimerIdType$

$timerKind : TimerKindType$

$timerState : TimerStateType$

$timerWheelRevolutions : \mathbb{N}$

$timerWheelRevolutionsLeft : \mathbb{N}$

$expirationCallbackAddr : MemoryAddressType$

$wheelSpokeIndex : TimerWheelSpokeIndexType$

$timerWheelRevolutionsLeft \leq timerWheelRevolutions$

$timerState = timerRunning \Rightarrow expirationCallbackAddr \neq nullAddress$

---*ThreadType*---
id : *ThreadIdType*
state : *ThreadStateType*
currentPriority : *ThreadPriorityType*
basePriority : *ThreadPriorityType*
atomicLevel : *AtomicLevelType*
builtinTimerId : *TimerIdType*
builtinCondvarId : *CondvarIdType*
waitingOnCondvarId : *CondvarIdType*
waitingOnMutexId : *MutexIdType*
ownedMutexes : iseq *ValidMutexIdType*
savedStackPointer : *MemoryAddressType*
stack : *AddressRangeType*
stackSavedCpuContext : *CpuRegistersType*
privilegedNestingCounter : $\mathbb{N}$
timeSliceLeftUs : $\mathbb{N}$

$state \neq threadNotCreated \Rightarrow$
$\quad (id \neq invalidThreadId \wedge$
$\quad builtinTimerId \neq invalidTimerId \wedge$
$\quad builtinCondvarId \neq invalidCondvarId \wedge$
$\quad basePriority \neq invalidThreadPriority \wedge$
$\quad currentPriority \neq invalidThreadPriority \wedge$
$\quad savedStackPointer \in zAddressRange(stack))$

$currentPriority \geq basePriority$

$state = threadBlockedOnCondvar \Leftrightarrow$
$\quad waitingOnCondvarId \neq invalidCondvarId$

$state = threadBlockedOnMutex \Leftrightarrow$
$\quad waitingOnMutexId \neq invalidMutexId$

$waitingOnCondvarId \neq invalidCondvarId \Rightarrow$
$\quad waitingOnMutexId = invalidMutexId$

$waitingOnMutexId \neq invalidMutexId \Rightarrow$
$\quad waitingOnCondvarId = invalidCondvarId$

$waitingOnMutexId \notin$ ran *ownedMutexes*

$\_$ *CondvarType* $_____$

id : *CondvarIdType*
wakeupAtomicLevel : *AtomicLevelType*
wakeupMutexId : *MutexIdType*
waitingThreadsQueue : *ThreadPriorityQueueType*

$_____$

wakeupAtomicLevel $\neq$ *atomicLevelNone* $\Rightarrow$ *wakeupMutexId* = *invalidMutexId*

$\_$ *MutexType* $_____$

id : *MutexIdType*
ownerThreadId : *ThreadIdType*
recursiveCount : $\mathbb{N}$
ceilingPriority : *ThreadPriorityType*
waitingThreadsQueue : *ThreadPriorityQueueType*

$_____$

waitingThreadsQueue.waitingThreadsCount $\neq 0 \Rightarrow$
    ownerThreadId $\neq$ *invalidThreadId*

ceilingPriority $\neq$ *invalidThreadPriority* $\Rightarrow$
$(\forall\, p : ValidThreadPriorityType \mid$
    waitingThreadsQueue.threadQueues$(p) \neq \emptyset \bullet$
    $p \leq ceilingPriority)$

3.2   HiRTOS Boot-time Initialization

When `HiRTOS.Initialize` is called for each CPU core, the idle thread and the tick timer thread for that CPU are created, but the thread scheduler is not started yet:

$\_$ *HiRtosInitialize* $_____$

HiRtos$'$
HiRtosCpuInstanceInitialize
cpuId? : *CpuIdType*

$_____$

cpuId$'$ = *cpuId*?

θ*HiRtosCpuInstanceType*$'$ = *rtosCpuInstances*$'($*cpuId*?$)$

interruptNestingLevelStack$'$.*zCpuId* = *cpuId*?

HiRtosCpuInstanceInitialize ─────────────────────────────

HiRtosCpuInstanceElaboration

$idleThreadId' \neq invalidThreadId$

$tickTimerThreadId' \neq invalidThreadId$

$tickTimerThreadId' \neq idleThreadId'$

dom $threads' = \{\, idleThreadId', tickTimerThreadId' \,\}$

dom $condvars' =$
$\{\, (threads'(idleThreadId')).builtinCondvarId,$
$(threads'(tickTimerThreadId')).builtinCondvarId \,\}$

dom $timers' =$
$\{\, (threads'(idleThreadId')).builtinTimerId,$
$(threads'(tickTimerThreadId')).builtinTimerId \,\}$

$zCpuInterruptMaskingState' = cpuInterruptsEnabled$

$zCpuPrivilege' = cpuUnprivileged$

$zMemoryProtectionState' = memoryProtectionOn$

$runnableThreadsQueue'.threadQueues(min\ ValidThreadPriorityType)$
$\quad = \langle idleThreadId' \rangle$

$runnableThreadsQueue'.threadQueues(max\ ValidThreadPriorityType)$
$\quad = \langle tickTimerThreadId' \rangle$

$\forall\, p : ValidThreadPriorityType \setminus$
$\qquad \{\, min\ ValidThreadPriorityType, max\ ValidThreadPriorityType \,\} \bullet$
$\quad runnableThreadsQueue'.threadQueues(p) = \emptyset$

HiRtosCpuInstanceElaboration

$HiRtosCpuInstanceType'$
$InterruptNestingLevelStackElaboration$

$threadSchedulerState' = threadSchedulerStopped$

$currentThreadId' = invalidThreadId$

$currentAtomicLevel' = atomicLevelNone$

$currentCpuExecutionMode' = cpuExecutingResetHandler$

$idleThreadId' = invalidThreadId$

$tickTimerThreadId' = invalidThreadId$

$threads' = \emptyset$

$condvars' = \emptyset$

$timers' = \emptyset$

$timerTicksSinceBoot' = 0$

InterruptNestingLevelStackElaboration

$InterruptNestingLevelStackType'$
$InterruptNestingLevelElaboration$

$currentInterruptNestingCounter' = 1$

$\forall x : ActiveInterruptNestingCounterType \bullet$
$\quad interruptNestingLevels'(x) = \theta InterruptNestingLevelType'$

InterruptNestingLevelElaboration

$InterruptNestingLevelType'$

$interruptId' = invalidInterruptId$

$interruptNestingCounter' = 0$

$savedStackPointer' = nullAddress$

$atomicLevel' = atomicLevelNone$

TimerWheelElaboration _____

TimerWheelType$'$

---

ran *wheelSpokesHashTable$'$* = { $\emptyset$ }

currentWheelSpokeIndex$'$ =
    min ValidTimerWheelSpokeIndexType

3.3 HiRTOS Callable Services

3.3.1 *HiRTOS* Threads Operations

Create a new thread

A thread can be created by calling `HiRTOS.Thread.Create_Thread`. Threads are allocated from the pool of thread objects of the calling CPU:

CreateThread _____

Δ *HiRtosCpuInstanceType*
InitializeNewThread
priority? : *ValidThreadPriorityType*

---

threadId! $\notin$ dom *threads*

condvarId! $\notin$ dom *condvars*

timerId! $\notin$ dom *timers*

threadId! $\in$ dom *threads$'$*

condvarId! $\in$ dom *condvars$'$*

timerId! $\in$ dom *timers$'$*

threads$'$(*threadId*!) = θ *ThreadType$'$*

runnableThreadsQueue$'$.threadQueues(*priority*?) =
    runnableThreadsQueue.threadQueues(*priority*?) $\frown$ $\langle$*threadId*!$\rangle$

```
┌─ InitializeNewThread ──────────────────────────────────────────
│ ThreadType′
│ condvarId! : ValidCondvarIdType
│ timerId! : ValidTimerIdType
│ threadId! : ValidThreadIdType
├────────────────────────────────────────────────────────────────
│ threadId! ≠ invalidThreadId
│
│ id′ = threadId!
│
│ builtinTimerId′ = timerId!
│
│ builtinCondvarId′ = condvarId!
│
│ state′ = threadRunnable
│
│ atomicLevel′ = atomicLevelNone
└────────────────────────────────────────────────────────────────
```

```
┌─ ThreadPriorityQueueInitialize ───────────────────────────────
│ ThreadPriorityQueueType′
├────────────────────────────────────────────────────────────────
│ ∀ p : ValidThreadPriorityType •
│     threadQueues′(p) = ∅
└────────────────────────────────────────────────────────────────
```

```
┌─ DequeueHighestPriorityThread ────────────────────────────────
│ ΔThreadPriorityQueueType
│ highestPriorityThreadId! : ValidThreadIdType
├────────────────────────────────────────────────────────────────
│ (let highestPrio ==
│     max (dom(threadQueues ▷ ∅)) •
│     highestPriorityThreadId! = head (threadQueues(highestPrio)) ∧
│     threadQueues′(highestPrio) = tail (threadQueues(highestPrio)))
└────────────────────────────────────────────────────────────────
```

3.3.2   *HiRTOS* Mutex Operations

Create a new mutex

A mutex can be created by calling `HiRTOS.Mutex.Create`. Mutexes are allocated from the pool of mutex objects of the calling CPU:

```
┌─ CreateMutex ────────────────────────────────
│ ΔHiRtosCpuInstanceType
│ InitializeNewMutex
├──────────────────────────────────────────────
│ mutexId! ∉ dom mutexes
│
│ mutexId! ∈ dom mutexes'
│
│ θMutexType' = mutexes'(mutexId!)
└──────────────────────────────────────────────
```

```
┌─ InitializeNewMutex ─────────────────────────
│ MutexType'
│ ceilingPriority? : ThreadPriorityType
│ mutexId! : ValidMutexIdType
├──────────────────────────────────────────────
│ mutexId! ≠ invalidMutexId
│
│ id' = mutexId!
│
│ ownerThreadId' = invalidThreadId
│
│ recursiveCount' = 0
│
│ ceilingPriority' = ceilingPriority?
│
│ ∀ p : ValidThreadPriorityType •
│       waitingThreadsQueue'.threadQueues(p) = ∅
└──────────────────────────────────────────────
```

If *ceilingPriority?* is *invalidThreadPriority* that means that the mutex follows the priority inheritance protocol. Otherwise, it follows the priority ceiling protocol. In the priority inheritance protocol, if the thread trying to acquire a busy mutex has higher priority than the thread currently owning the mutex, the owning thread gets its priority raised to the priority of the waiting thread. In the priority ceiling protocol, when a thread acquires a mutex, if the mutex's ceiling priority is higher than the thread's priority, the thread gets its priority raised to the ceiling priority.

The *CreatedMutexMutableOperation* schema below is used in the specifications of all the mutable operations that can be performed on mutexes that were previously created by a call to `HiRTOS.Mutex.Create`:

CreatedMutexMutableOperation ─────────────────────────────

$\Delta HiRtos$
$\Delta HiRtosCpuInstanceType$
$\Delta MutexType$
$cpuId? : CpuIdType$
$mutexId? : ValidMutexIdType$

─────

$\theta HiRtosCpuInstanceType = rtosCpuInstances(cpuId?)$

$\theta HiRtosCpuInstanceType' = rtosCpuInstances'(cpuId?)$

$\theta MutexType = mutexes(mutexId?)$

$\theta MutexType' = mutexes'(mutexId?)$

Acquire a mutex

A thread acquires a mutex by calling `HiRTOS.Mutex.Acquire`, according to the contract specified by the *AcquireMutex* schema:

AcquireAvailableMutex ─────────────────────────────

CreatedMutexMutableOperation

─────

$ownerThreadId = invalidThreadId \lor$
$(ownerThreadId = currentThreadId \Rightarrow$
$recursiveCount' = recursiveCount + 1)$

$ownerThreadId' = currentThreadId$

$(ceilingPriority \neq invalidThreadPriority \land$
$(threads(currentThreadId)).currentPriority <$
$\quad\quad ceilingPriority) \Rightarrow$
$(threads'(currentThreadId)).currentPriority =$
$\quad\quad ceilingPriority$

$(threads'(currentThreadId)).ownedMutexes =$
$\quad\quad (threads(currentThreadId)).ownedMutexes \frown \langle mutexId? \rangle$

WaitOnUnavailableMutex

CreatedMutexMutableOperation
DequeueHighestPriorityThread

$runnableThreadsQueue = \theta\,ThreadPriorityQueueType$

$runnableThreadsQueue' = \theta\,ThreadPriorityQueueType'$

$ownerThreadId \neq invalidThreadId$

$currentThreadId \neq ownerThreadId$

$currentThreadId' \neq currentThreadId$

(**let** $oldCurrentPriority ==$
    $(threads(currentThreadId)).currentPriority \bullet$
$(ceilingPriority = invalidThreadPriority \land$
$oldCurrentPriority >$
    $(threads(ownerThreadId)).currentPriority) \Rightarrow$
$(threads'(ownerThreadId)).currentPriority =$
    $oldCurrentPriority$
$\land$
$currentThreadId' \neq ownerThreadId \Rightarrow$
$((threads'(currentThreadId')).currentPriority \geq$
    $(threads'(ownerThreadId)).currentPriority \lor$
$(threads'(ownerThreadId)).state \in \{threadBlockedOnMutex, threadBlockedOnCondvar\})$
$\land$
$(threads'(currentThreadId)).state = threadBlockedOnMutex$
$\land$
$currentThreadId \in$
    $\mathrm{ran}(waitingThreadsQueue'.threadQueues(oldCurrentPriority)))$

$ceilingPriority \neq invalidThreadPriority \Rightarrow$
$((threads(currentThreadId)).currentPriority \leq$
    $(threads'(ownerThreadId)).currentPriority \land$
$(threads'(ownerThreadId)).state \in \{threadBlockedOnMutex, threadBlockedOnCondvar\})$

$currentThreadId' \neq currentThreadId$

$currentThreadId' = highestPriorityThreadId!$

$AcquireMutex \;\widehat{=}$
    $AcquireAvailableMutex \lor WaitOnUnavailableMutex$

Release a mutex

A thread releases a mutex by calling `HiRTOS.Mutex.Release`, according to the contract specified by the *ReleaseMutex* schema:

ReleaseMutex _____
CreatedMutexMutableOperation
DequeueHighestPriorityThread

$waitingThreadsQueue = \theta\,ThreadPriorityQueueType$

$waitingThreadsQueue' = \theta\,ThreadPriorityQueueType'$

$mutexId? = head\,(threads(currentThreadId)).ownedMutexes$

$(threads'(currentThreadId)).ownedMutexes = tail\,(threads(currentThreadId)).ownedMute$

$(\textbf{let}\ p == (threads(highestPriorityThreadId!)).currentPriority\ \bullet$
$\qquad runnableThreadsQueue'.threadQueues(p) =$
$\qquad\qquad runnableThreadsQueue.threadQueues(p) \frown \langle highestPriorityThreadId! \rangle)$

3.3.3   *HiRTOS* Condition Variable Operations

Create a new condition variable

A condition variable can be created by calling `HiRTOS.Condvar.Create`. Condvars are allocated from the pool of condvar objects of the calling CPU:

CreateCondvar _____
$\Delta HiRtosCpuInstanceType$
InitializeNewCondvar

$condvarId! \notin \mathrm{dom}\,condvars$

$condvarId! \in \mathrm{dom}\,condvars'$

$condvars'(condvarId!) = \theta\,CondvarType'$

$$
\begin{array}{|l}
\hline
\_\mathit{InitializeNewCondvar} _____ \\
\mathit{CondvarType'} \\
\mathit{ThreadPriorityQueueInitialize} \\
\mathit{condvarId!} : \mathit{ValidCondvarIdType} \\
\hline
\mathit{waitingThreadsQueue'} = \theta\,\mathit{ThreadPriorityQueueType'} \\[2pt]
\mathit{condvarId!} \neq \mathit{invalidCondvarId} \\[2pt]
\mathit{id'} = \mathit{condvarId!} \\[2pt]
\mathit{wakeupAtomicLevel'} = \mathit{atomicLevelNone} \\[2pt]
\mathit{wakeupMutexId'} = \mathit{invalidMutexId} \\
\hline
\end{array}
$$

The *CreatedCondvarMutableOperation* schema below is used in the specifications of all the mutable operations that can be performed on condition variables that were previously created by a call to `HiRTOS.Condvar.Create`:

$$
\begin{array}{|l}
\hline
\_\mathit{CreatedCondvarMutableOperation} _____ \\
\Delta\mathit{HiRtos} \\
\Delta\mathit{HiRtosCpuInstanceType} \\
\Delta\mathit{CondvarType} \\
\mathit{cpuId?} : \mathit{CpuIdType} \\
\mathit{condvarId?} : \mathit{ValidCondvarIdType} \\
\hline
\mathit{rtosCpuInstances}(\mathit{cpuId?}) = \theta\,\mathit{HiRtosCpuInstanceType} \\[2pt]
\mathit{rtosCpuInstances'}(\mathit{cpuId?}) = \theta\,\mathit{HiRtosCpuInstanceType'} \\[2pt]
\mathit{condvars}(\mathit{condvarId?}) = \theta\,\mathit{CondvarType} \\[2pt]
\mathit{condvars'}(\mathit{condvarId?}) = \theta\,\mathit{CondvarType'} \\
\hline
\end{array}
$$

Wait on a condition variable

A thread waits on a condition variable by calling `HiRTOS.Condvar.Wait`, according to the contract specified by the *WaitOnCondvar* schema:

$\quad$ *WaitOnCondvar* $\underline{\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad}$

$\quad$ *CreatedCondvarMutableOperation*
$\quad$ *DequeueHighestPriorityThread*
$\quad$ *mutexId?* : *ValidMutexIdType*

$\overline{}$

$\quad$ *runnableThreadsQueue* $= \theta\, ThreadPriorityQueueType$

$\quad$ *runnableThreadsQueue′* $= \theta\, ThreadPriorityQueueType′$

$\quad$ *mutexId?* $\in \mathrm{ran}(threads(currentThreadId)).ownedMutexes$

$\quad$ *mutexId?* $\notin \mathrm{ran}(threads′(currentThreadId)).ownedMutexes$

$\quad$ *wakeupMutexId′* $= mutexId?$

$\quad$ (**let** $p == (threads(currentThreadId)).currentPriority \bullet$
$\qquad waitingThreadsQueue′.threadQueues(p) =$
$\qquad\qquad waitingThreadsQueue.threadQueues(p) \frown \langle currentThreadId \rangle)$

$\quad$ *currentThreadId′* $\neq currentThreadId$

$\quad$ *currentThreadId′* $= highestPriorityThreadId!$

Signal a condition variable

A thread signals a condition variable by calling `HiRTOS.Condvar.Signal`, according to the contract specified by the *SignalCondvar* schema. Signaling a condition variable wakes up the highest priority thread waiting on the condition variable.

$\quad$ *SignalCondvar* $\underline{\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad}$

$\quad$ *CreatedCondvarMutableOperation*
$\quad$ *DequeueHighestPriorityThread*

$\overline{}$

$\quad$ *waitingThreadsQueue* $= \theta\, ThreadPriorityQueueType$

$\quad$ *waitingThreadsQueue′* $= \theta\, ThreadPriorityQueueType′$

$\quad$ *wakeupMutexId* $\in \mathrm{ran}(threads′(highestPriorityThreadId!)).ownedMutexes$

$\quad$ (**let** $p == (threads(highestPriorityThreadId!)).currentPriority \bullet$
$\qquad runnableThreadsQueue′.threadQueues(p) =$
$\qquad\qquad runnableThreadsQueue.threadQueues(p) \frown \langle highestPriorityThreadId! \rangle)$

Broadcast on a condition variable

A thread broadcasts on a condition variable by calling `HiRTOS.Condvar.Broadcast`, according to the contract specified by the *BroadcastCondvar* schema. Broadcasting on a condition variable wakes up all threads waiting on the condition variable.

BroadcastCondvar
CreatedCondvarMutableOperation

$(\forall\, p : ValidThreadPriorityType\,|$
$\quad waitingThreadsQueue.threadQueues(p) \neq \emptyset \bullet$
$\quad waitingThreadsQueue'.threadQueues(p) = \emptyset \,\wedge$
$\quad runnableThreadsQueue'.threadQueues(p) =$
$\qquad runnableThreadsQueue.threadQueues(p)^\frown$
$\qquad waitingThreadsQueue.threadQueues(p))$

3.3.4 *HiRTOS* Software Timer Operations

Create a new software timer

A software timer can be created by calling `HiRTOS.Timer.Create`. Timers are allocated from the pool of timer objects of the calling CPU:

CreateTimer
$\Delta HiRtosCpuInstanceType$
InitializeNewTimer

$timerId! \notin \mathrm{dom}\, timers$

$timerId! \in \mathrm{dom}\, timers'$

$timers'(timerId!) = \theta\, TimerType'$

InitializeNewTimer
$TimerType'$
$timerId! : ValidTimerIdType$

$timerId! \neq invalidTimerId$

$id' = timerId!$

$timerKind' = oneShotTimer$

$timerState' = timerStopped$

$timerWheelRevolutions' = 0$

$timerWheelRevolutionsLeft' = 0$

$expirationCallbackAddr' = nullAddress$

$wheelSpokeIndex' = invalidTimerWheelSpokeIndex$

The *CreatedTimerMutableOperation* schema below is used in the specifications

of all the mutable operations that can be performed on software timers that were previously created by a call to `HiRTOS.Timer.Create`:

```
_CreatedTimerMutableOperation _____
ΔHiRtos
ΔHiRtosCpuInstanceType
ΔTimerType
cpuId? : CpuIdType
timerId? : ValidTimerIdType
_____
rtosCpuInstances(cpuId?) = θHiRtosCpuInstanceType

rtosCpuInstances′(cpuId?) = θHiRtosCpuInstanceType′

timers(timerId?) = θTimerType

timers′(timerId?) = θTimerType′
```

Start a software timer

A software timer is started by calling `HiRTOS.Timer.Start_Timer`, according to the contract specified by the *StartTimer* schema:

```
_StartTimer _____
CreatedTimerMutableOperation
expirationTimeUs? : ℕ
expirationCallbackAddr? : MemoryAddressType
timerKind? : TimerKindType
_____
expirationCallbackAddr? ≠ nullAddress

expirationCallbackAddr′ = expirationCallbackAddr?

timerKind′ = timerKind?

(let expirationTimeTicks == expirationTimeUs? div TickTimerPeriodUs •
    wheelSpokeIndex′ =
        (timerWheel.currentWheelSpokeIndex + expirationTimeTicks) mod numTimerV
    timerWheelRevolutions′ = expirationTimeTicks div numTimerWheelSpokes ∧
    timerWheelRevolutionsLeft′ = timerWheelRevolutions′ ∧
    timerWheel′.wheelSpokesHashTable(wheelSpokeIndex′) =
        timerWheel.wheelSpokesHashTable(wheelSpokeIndex′) ∪ { timerId? })
```

Stop a software timer

A software timer is stopped by calling `HiRTOS.Timer.Stop_Timer`, according to the contract specified by the *StopTimer* schema:

```
┌─ StopTimer ──────────────────────────────────────────────────
│ CreatedTimerMutableOperation
│ ─────────────────────────────────────────────────────────────
│ expirationCallbackAddr′ = nullAddress
│
│ timerWheel′.wheelSpokesHashTable(wheelSpokeIndex) =
│      timerWheel.wheelSpokesHashTable(wheelSpokeIndex) \ { timerId? }
└──────────────────────────────────────────────────────────────
```

Pointer-less Linked list

Thread queues, mutex lists and timer sets are refined as pointer-less linked lists. Pointer-less linked lists use array indices instead of physical pointers. The *GenericLinkedLists* schema represents a generic pointer-less linked list in which the links that chain the list toegether are values of type *ElementIdType* which corresponds to indices of some object array.

```
┌─ GenericLinkedList[ElementIdType] ──────────────────────────
│ head : ElementIdType
│ tail : ElementIdType
│ length : ℕ
│ nextElement : ElementIdType ⇸↠ ElementIdType
│ prevElement : ElementIdType ⇸↠ ElementIdType
│ ─────────────────────────────────────────────────────────────
│ length ≤ #ElementIdType
│
│ length = #(dom nextElement)
│
│ dom nextElement = dom prevElement
│
│ dom nextElement = dom prevElement
│
│ length ≤ 1 ⇔ head = tail
│
│ length ≥ 2 ⇔
│      (nextElement^{length−1}(head) = tail ∧ prevElement^{length−1}(tail) = head)
│
│ nextElement^+ ∩ id ElementIdType = ∅
│
│ prevElement^+ ∩ id ElementIdType = ∅
│
│ ∀ x : dom nextElement •
│      (x ≠ tail ⇒ prevElement(nextElement(x)) = x) ∧
│      (x ≠ head ⇒ nextElement(prevElement(x)) = x)
└──────────────────────────────────────────────────────────────
```

ThreadQueueType which is an injective sequence is refined as the *RefinedThreadQueueType* schema:

RefinedThreadQueueType ────────────────────────────

$GenericLinkedList[ThreadIdType]$

───────

$nextElement(tail) = invalidThreadId$

$prevElement(tail) = invalidThreadId$

$length = 0 \Leftrightarrow head = invalidThreadId$

$tail = invalidThreadId \Leftrightarrow head = invalidThreadId$
───────

MutexListType which is also an injective sequencce is refined as the *RefinedMutexListType* schema:

RefinedMutexListType ────────────────────────────

$GenericLinkedList[MutexIdType]$

───────

$nextElement(tail) = invalidMutexId$

$prevElement(tail) = invalidMutexId$

$length = 0 \Leftrightarrow head = invalidMutexId$

$tail = invalidMutexId \Leftrightarrow head = invalidMutexId$
───────

The set of timers associated with each timerwheel spoke is refined as the *TimerListType* schema:

TimerListType ────────────────────────────

$GenericLinkedList[TimerIdType]$

───────

$nextElement(tail) = invalidTimerId$

$prevElement(tail) = invalidTimerId$

$length = 0 \Leftrightarrow head = invalidTimerId$

$tail = invalidTimerId \Leftrightarrow head = invalidTimerId$
───────

3.4 HiRTOS Thread Context Switching

3.4.1 Asynchronous Thread Context Switch

In HiRTOS, thread preemption is implemented by invoking the thread scheduler on the exit path of an interrupt handler. When an interrupt fires while a thread is running, the executing thread's CPU context is saved on thread's stack by the interrupt handler prolog. Then, before calling the actual interrupt handler, the stack is switched to the interrupt handling stack. After the interrupt handler returns, the

interrupt handler epilog invokes the HiRTOS thread scheduler, to select the highest priority runnable thread. Then, if the newly selected thread is different from the one that was running before the interrupt, the extended context of the old thread is saved and the extended context of the new thread is restored.
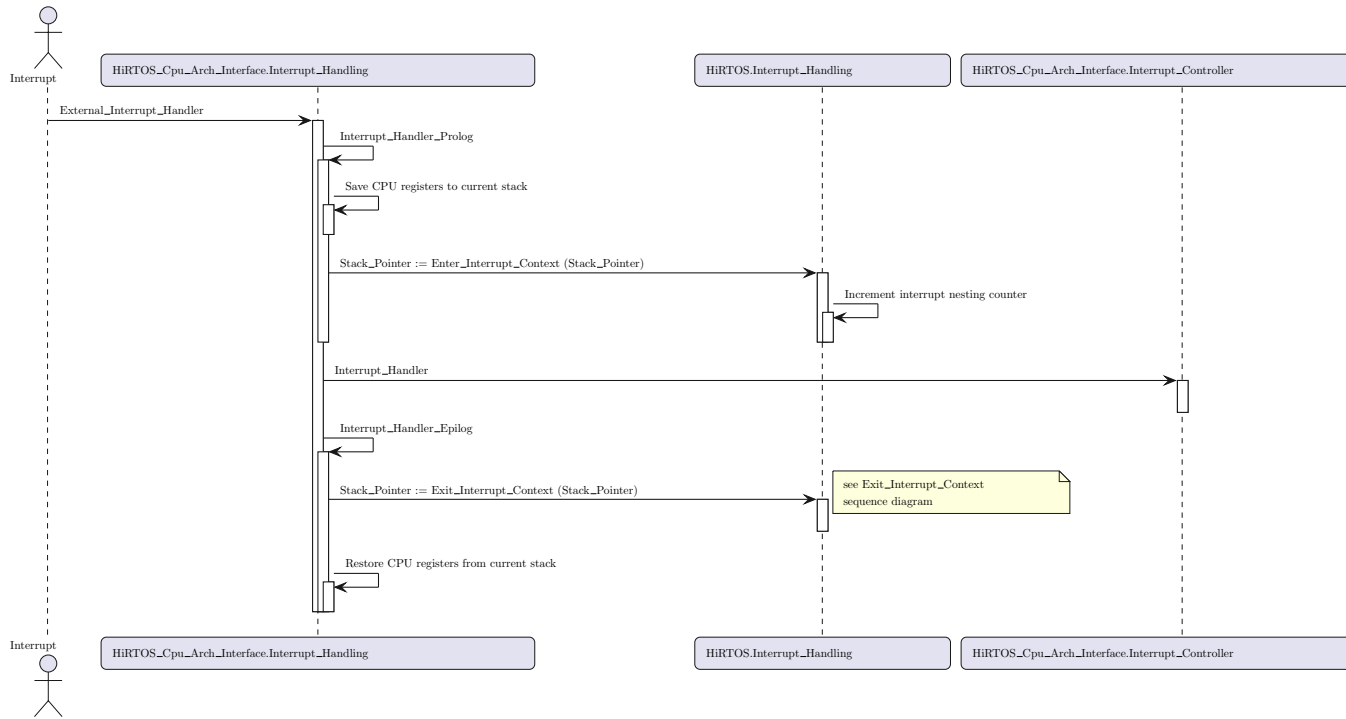

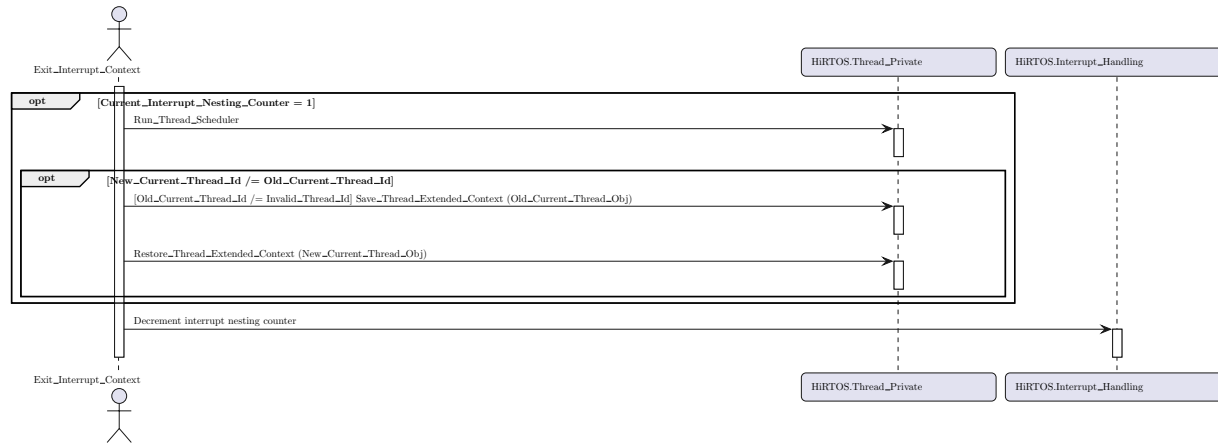Figure 3.1: HiRTOS Asynchronous Thread Context Switch - part 1

Figure 3.2: HiRTOS Asynchronous Thread Context Switch - part 2

3.4.2   Synchronous Thread Context Switch

In HiRTOS, a synchronous thread context switch occurs in the following cases:

- When a thread calls `HiRTOS.Condvar.Wait`

- When a thread calls `HiRTOS.Condvar.Signal` or `HiRTOS.Condvar.Broadcast`, and there are threads waiting on the condition variable

- When a thread calls `HiRTOS.Mutex.Acquire` and the mutex is not available

- When a thread calls `HiRTOs.Mutex.Release` and there are threads waiting to acquire the mutex

- When a thread calls `HiRTOS.Thread.Thread_Delay_Until` (calls `HiRTOS.Condvar.Wait`)

- When a thread calls `HiRTOS.Thread.Suspend_Current_Thread`

- When a thread calls `HiRTOS.Thread.Resume_Thread`

- When a thread calls `HiRTOS.Restore_Atomic_Level` and the old atomic level is `Atomic_Level_None`
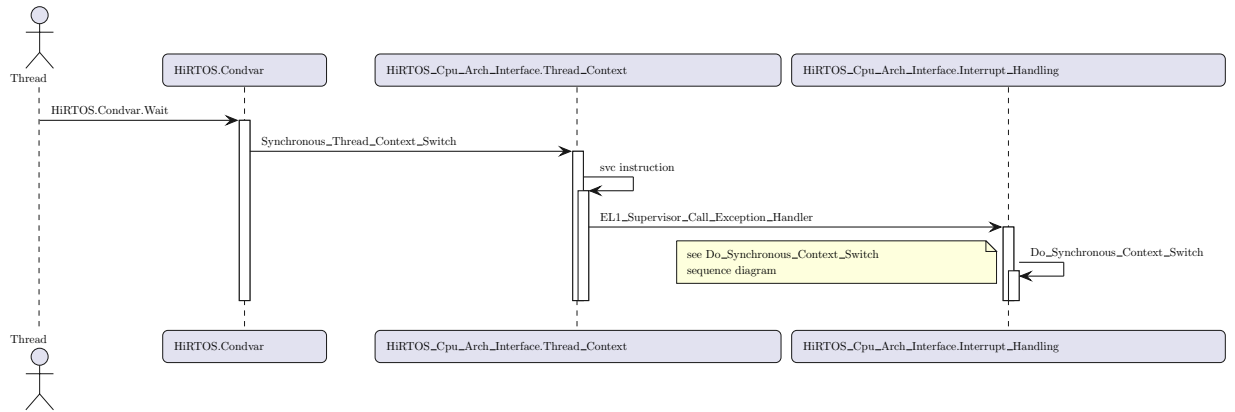
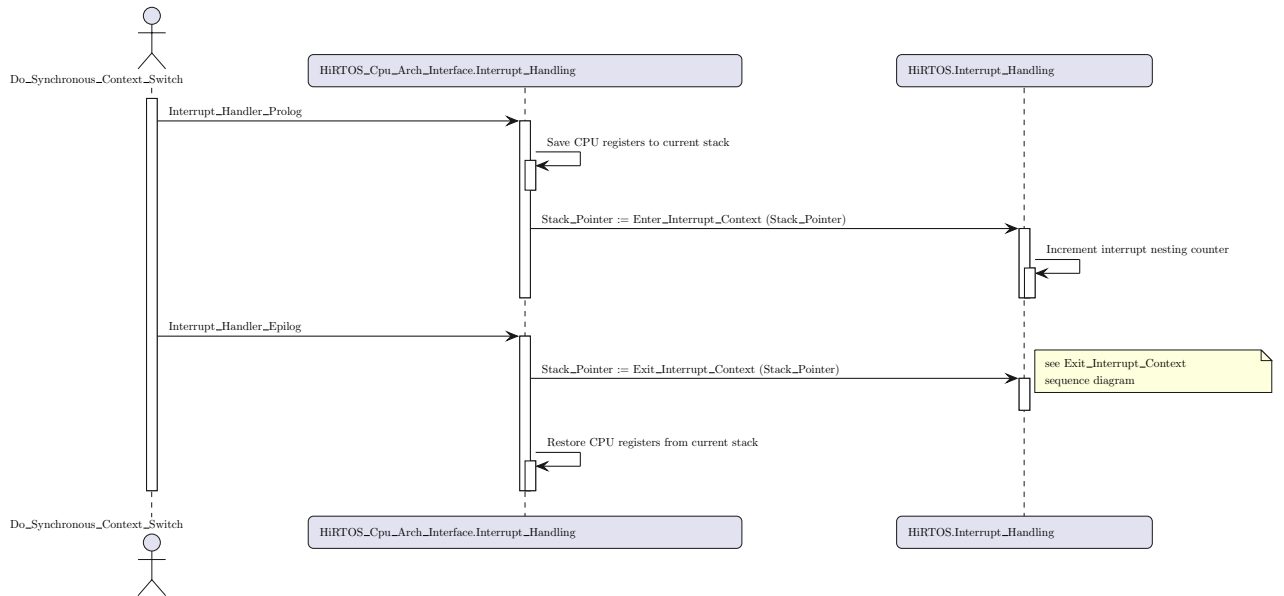Figure 3.3: HiRTOS Synchronous Thread Context Switch - part 1

Figure 3.4: HiRTOS Synchronous Thread Context Switch - part 2

A special case is the initial thread context switch, which occurs when the HiRTOS thread scheduler is started, when the application calls `HiRTOS.Start_Thread_Scheduler`.

Figure 3.5: HiRTOS Initial Thread Context Switch - part 1
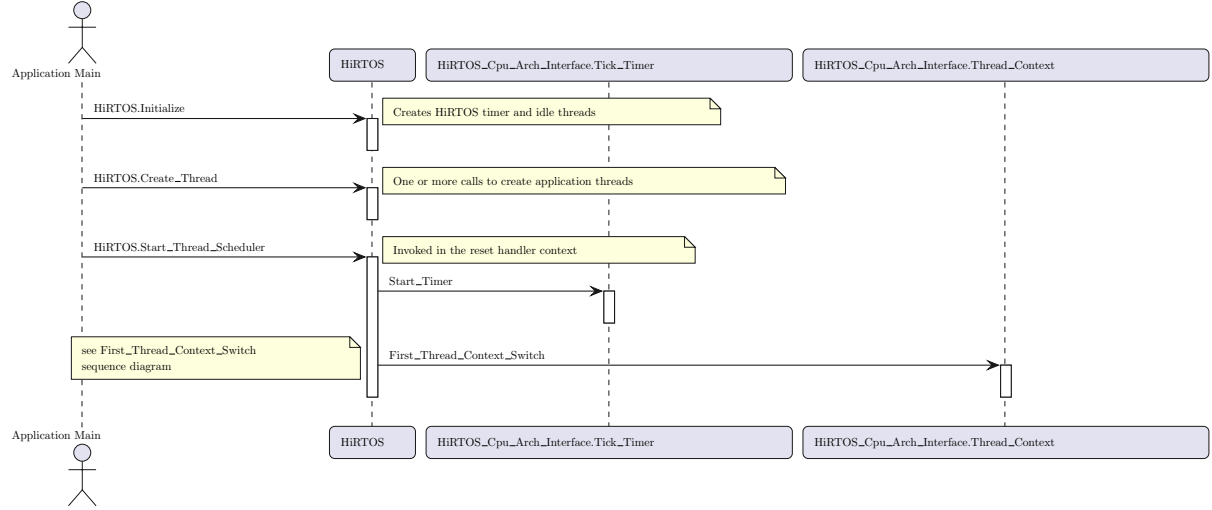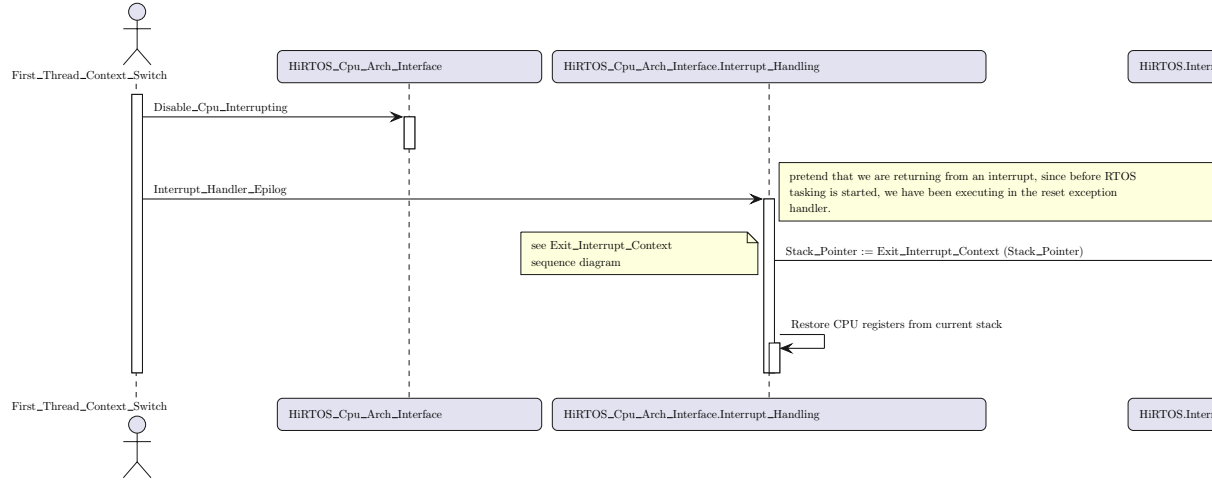

Figure 3.6: HiRTOS Initial Thread Context Switch - part 2

3.4.3   TLA+/PlusCal Model of the HiRTOS Thread Scheduler

The HiRTOS thread scheduler has been modeled in TLA+/PlusCal [15, 16]. The HiRTOS TLA+/PlusCal model can be found here. The following safety invariants were verified with the TLC model checker [17]:

- There can be at most only one "running" thread

```
HiRTOS.Interrupts_Enabled =>
    IF HiRTOS.Current_Thread_Id /= "Invalid_Thread_Id" THEN
        /\ Cardinality({ t \in Threads : Thread_Objects[t].State = "Running" }) = 1
        /\ HiRTOS.Current_Thread_Id =
            CHOOSE t \in Threads : Thread_Objects[t].State = "Running"
    ELSE
        { t \in Threads : Thread_Objects[t].State = "Running" } = {}
```

- The running thread is not in any queue

```
(HiRTOS.Interrupts_Enabled /\
 HiRTOS.Current_Thread_Id /= "Invalid_Thread_Id") =>
    (/\ Thread_Objects[HiRTOS.Current_Thread_Id].State = "Running"
     /\ Thread_Objects[HiRTOS.Current_Thread_Id].Waiting_On_Condvar_Id =
            "Invalid_Condvar_Id"
     /\ Thread_Objects[HiRTOS.Current_Thread_Id].Waiting_On_Mutex_Id =
            "Invalid_Mutex_Id"
     /\ ~Is_Thread_In_Priority_Queue(HiRTOS.Runnable_Threads_Queue,
                                        HiRTOS.Current_Thread_Id)
     /\ \A m \in Mutexes :
            ~Is_Thread_In_Priority_Queue(Mutex_Objects[m].Waiting_Threads_Queue,
                                            HiRTOS.Current_Thread_Id)
     /\ \A cv \in Condvars :
            ~Is_Thread_In_Priority_Queue(Condvar_Objects[cv].Waiting_Threads_Queue,
                                            HiRTOS.Current_Thread_Id)
    )
```

- All Runnable threads are in the Runnable threads queue and no other queue

```
HiRTOS.Interrupts_Enabled =>
\A t \in Threads : Thread_Objects[t].State = "Runnable" =>
    /\ Thread_Objects[t].Waiting_On_Condvar_Id = "Invalid_Condvar_Id"
    /\ Thread_Objects[t].Waiting_On_Mutex_Id = "Invalid_Mutex_Id"
    /\ Is_Thread_In_Priority_Queue(HiRTOS.Runnable_Threads_Queue, t)
    /\ Is_Thread_In_Priority_Queue_In_Only_One_Queue(
            HiRTOS.Runnable_Threads_Queue, t)
    /\ \A m \in Mutexes :
            ~Is_Thread_In_Priority_Queue(Mutex_Objects[m].Waiting_Threads_Queue, t)
    /\ \A cv \in Condvars :
            ~Is_Thread_In_Priority_Queue(
                Condvar_Objects[cv].Waiting_Threads_Queue, t)
```

- Each thread blocked on a mutex in only one mutex's wait queue and no other queue

```
HiRTOS.Interrupts_Enabled =>
\A t \in Threads :
LET
    thread_obj == Thread_Objects[t]
```

```
IN
    thread_obj.State = "Blocked_On_Mutex" =>
    /\ thread_obj.Waiting_On_Mutex_Id /= "Invalid_Mutex_Id"
    /\ thread_obj.Waiting_On_Condvar_Id = "Invalid_Condvar_Id"
    /\ (LET
            mutex_obj == Mutex_Objects[thread_obj.Waiting_On_Mutex_Id]
        IN
            /\ Is_Thread_In_Priority_Queue(mutex_obj.Waiting_Threads_Queue,
            /\ Is_Thread_In_Priority_Queue_In_Only_One_Queue(
                    mutex_obj.Waiting_Threads_Queue, t))
    /\ ~Is_Thread_In_Priority_Queue(HiRTOS.Runnable_Threads_Queue, t)
    /\ \A m \in Mutexes \ { thread_obj.Waiting_On_Mutex_Id } :
            ~Is_Thread_In_Priority_Queue(
                Mutex_Objects[m].Waiting_Threads_Queue, t)
    /\ \A cv \in Condvars :
            ~Is_Thread_In_Priority_Queue(
                Condvar_Objects[cv].Waiting_Threads_Queue, t)
```

- Each thread blocked on a condvar is in only one condvarś wait queue and no other queue

```
HiRTOS.Interrupts_Enabled =>
\A t \in Threads :
LET
    thread_obj == Thread_Objects[t]
IN
    thread_obj.State = "Blocked_On_Condvar" =>
    /\ thread_obj.Waiting_On_Condvar_Id /= "Invalid_Condvar_Id"
    /\ thread_obj.Waiting_On_Mutex_Id = "Invalid_Mutex_Id"
    /\ (LET
            condvar_obj == Condvar_Objects[thread_obj.Waiting_On_Condvar_Id]
        IN
            /\ Is_Thread_In_Priority_Queue(
                    condvar_obj.Waiting_Threads_Queue, t)
            /\ Is_Thread_In_Priority_Queue_In_Only_One_Queue(
                    condvar_obj.Waiting_Threads_Queue, t))
    /\ ~Is_Thread_In_Priority_Queue(HiRTOS.Runnable_Threads_Queue, t)
    /\ \A cv \in Condvars \ { thread_obj.Waiting_On_Condvar_Id } :
            ~Is_Thread_In_Priority_Queue(
                Condvar_Objects[cv].Waiting_Threads_Queue, t)
    /\ \A m \in Mutexes :
            ~Is_Thread_In_Priority_Queue(
                Mutex_Objects[m].Waiting_Threads_Queue, t)
```

- Each mutex that is currently owned by a thread must be in the list of mutexes owned by that thread

```
HiRTOS.Interrupts_Enabled =>
\A m \in Mutexes :
```

```
LET
    t == Mutex_Objects[m].Owner_Thread_Id
IN
    t /= "Invalid_Thread_Id" =>
        m \in Range(Thread_Objects[t].Owned_Mutexes)
```

- If a mutex is not owned by a thread, its wait queue should be empty

```
HiRTOS.Interrupts_Enabled =>
\A m \in Mutexes :
    Mutex_Objects[m].Owner_Thread_Id = "Invalid_Thread_Id" =>
        Is_Thread_Priority_Queue_Empty(Mutex_Objects[m].Waiting_Threads_Queue)
```

- The thread owning a mutex can never have lower priority than any thread waiting for the mutex

```
(HiRTOS.Interrupts_Enabled /\
 HiRTOS.Current_Thread_Id /= "Invalid_Thread_Id") =>
\A m \in Mutexes :
LET
    t == Mutex_Objects[m].Owner_Thread_Id
    prio_queue == Mutex_Objects[m].Waiting_Threads_Queue
IN
    (t /= "Invalid_Thread_Id" /\
     ~Is_Thread_Priority_Queue_Empty(prio_queue)) =>
    \A wt \in UNION { Range(q) : q \in Range(prio_queue) } :
        Thread_Objects[wt].State = "Blocked_On_Mutex" /\
        Thread_Objects[t].Current_Priority >= Thread_Objects[wt].Current_Priority
```

- A thread not owning any mutex and not waiting on a condvar always has its current priority set to its base priority

```
HiRTOS.Interrupts_Enabled =>
\A t \in Threads :
    (Thread_Objects[t].Owned_Mutexes = <<>> /\
     Thread_Objects[t].State /= "Blocked_On_Condvar") =>
    Thread_Objects[t].Current_Priority = Thread_Objects[t].Base_Priority
```

Also, the following liveness properties were verified with TLC:

- Interrupts are not disabled indefinitely

```
~HiRTOS.Interrupts_Enabled => <>HiRTOS.Interrupts_Enabled
```

- Every thread waiting to acquire a mutex, eventually gets it:

```
\A t \in Threads :
    Thread_Objects[t].State = "Waiting_On_Mutex" =>
    <>(Thread_Objects[t].State = "Runnable")
```

- Every thread waiting on a condvar is eventually awaken:

```
\A t \in Threads :
    Thread_Objects[t].State = "Waiting_On_Mutex" =>
    <>(Thread_Objects[t].State = "Runnable")
```

Chapter 4

HiRTOS Separation Kernel Z Specification

4.1 Separation Kernel Data Structures

4.1.1 Separation Kernel Configuration Parameters

Constants defined here represent compile-time configuration parameters for the *HiRTOS* Separation kernel.

$$
\begin{array}{|l}
maxNumPartitionsPerCpu : \mathbb{N}_1 \\
TickTimerPeriodUs : \mathbb{N}_1 \\
PartitionTimeSliceTicks : \mathbb{N}_1
\end{array}
$$

4.1.2 Separation Kernel Primitive Types

Below are the primitive types used in HiRTOS:

$[PartitionIdType]$
$\#PartitionIdType = maxNumPartitionsPerCpu + 1$

$$
\begin{array}{|l}
invalidPartitionId : PartitionIdType
\end{array}
$$

$ValidPartitionIdType ==$
$\qquad PartitionIdType \setminus \{invalidPartitionId\}$
$PartitionQueueType == \text{iseq } ValidPartitionIdType$
$PartitionStateType ::= partitionNotCreated \mid$
$\qquad\qquad\qquad partitionRunnable \mid partitionRunning \mid$
$\qquad\qquad\qquad partitionSuspended$
$PartitionSchedulerStateType ::=$
$\qquad partitionSchedulerStopped \mid partitionSchedulerRunning$
$AddressRangeType == MemoryAddressType \times MemoryAddressType$

4.1.3   Separation Kernel State Variables

The *HiRtosSeparationKernel* schema represents the internal data structures of the HiRTOS separation kernel.

HiRtosSeparationKernel _____

$separationKernelCpuInstances :$
$\quad ValidCpuIdType \nrightarrow\!\!\!\rightarrow SeparationKernelCpuInstanceType$
$zAllCreatedPartitionInstances : \mathbb{F}\ PartitionType$

$\#separationKernelCpuInstances \geq 1$

$\forall\ i : \mathrm{dom}\ separationKernelCpuInstances \bullet$
$\quad (separationKernelCpuInstances(i)).cpuId = i$

$zAllCreatedPartitionInstances =$
$\quad \bigcup \{\ i : ValidCpuIdType \bullet$
$\qquad \mathrm{ran}\ (separationKernelCpuInstances(i)).partitions\ \}$

$\bigcap \{\ i : ValidCpuIdType \bullet$
$\quad \mathrm{ran}\ (separationKernelCpuInstances(i)).partitions\ \} = \emptyset$

$\bigcap \{\ r : \bigcup \{\ p : zAllCreatedPartitionInstances \bullet p.savedHypervisorLevelMpuRegions\ \} \bullet$
$\quad zAddressRange(r)\ \} = \emptyset$

Per-CPU Separation Kernel Instance

The state variables and internal data structures of each per-CPU *HiRTOS* separation kernel instance are described below:

─── *SeparationKernelCpuInstanceType* ──────────────────
cpuId : *ValidCpuIdType*
interruptStack : *AddressRangeType*
partitionSchedulerState : *PartitionSchedulerStateType*
currentPartitionId : *PartitionIdType*
timerTicksSinceBoot : $\mathbb{N}$
runnablePartitionsQueue : *PartitionQueueType*
partitions : *ValidPartitionIdType* $\rightarrowtail\!\!\!\!\!\rightarrow$ *PartitionType*
zCurrentCpuContext : *CpuRegistersType*
zCurrentHypervisorLevelMpuRegions : $\mathbb{F}_1$ *AddressRangeType*
zCurrentSupervisorLevelMpuRegions : $\mathbb{F}$ *AddressRangeType*
zCurrentSupervisorLevelInterruptVectorTableAddr : *MemoryAddressType*
zCurrentSupervisorLevelInterruptsEnabled : $\mathbb{F}$ *ValidInterruptIdType*
zCurrentHighestInterruptPriorityDisabled : *InterruptPriorityType*
──────────────
zAddressRange(*interruptStack*) $\neq \emptyset$

currentPartitionId $\neq$ *invalidPartitionId* $\Rightarrow$
    currentPartitionId $\in$ dom *partitions* $\wedge$
    zCurrentHypervisorLevelMpuRegions =
        (*partitions*(*currentPartitionId*)).*savedHypervisorLevelMpuRegions* $\wedge$
    zCurrentSupervisorLevelInterruptVectorTableAddr =
        (*partitions*(*currentPartitionId*)).*savedInterruptVectorTableAddr*

$\forall\, p$: ran *partitions* $\bullet$
    $\bigcap \{\, i : p.savedHypervisorLevelMpuRegions \bullet zAddressRange(i) \,\} = \emptyset$

$\forall\, i$: *zCurrentSupervisorLevelMpuRegions* $\bullet$
    $\exists_1\, j$: *zCurrentHypervisorLevelMpuRegions* $\bullet$
        zAddressRange(*i*) $\subseteq$ *zAddressRange*(*j*)
──

─── *PartitionType* ───────────────────────────
id : *PartitionIdType*
failoverPartitionId : *PartitionIdType*
state : *PartitionStateType*
timeSliceLeftUs : $\mathbb{N}$
savedCpuContext : *CpuRegistersType*
savedHypervisorLevelMpuRegions : $\mathbb{F}_1$ *AddressRangeType*
savedSupervisorLevelMpuRegions : $\mathbb{F}$ *AddressRangeType*
savedInterruptVectorTableAddr : *MemoryAddressType*
savedInterruptsEnabled : $\mathbb{F}$ *InterruptIdType*
savedHighestInterruptPriorityDisabled : *InterruptPriorityType*
─────────────
state $\neq$ *partitionNotCreated* $\Rightarrow$
    id $\neq$ *invalidPartitionId*

failoverPartitionId $\neq$ *invalidPartitionId* $\Rightarrow$
    failoverPartitionId $\neq$ *id*

$\bigcap \{\, i : savedHypervisorLevelMpuRegions \bullet zAddressRange(i) \,\} = \emptyset$

$\emptyset \notin \{\, i : savedHypervisorLevelMpuRegions \bullet zAddressRange(i) \,\}$

$\forall\, i : savedSupervisorLevelMpuRegions \bullet$
    $\exists_1 j : savedHypervisorLevelMpuRegions \bullet$
        $zAddressRange(i) \subseteq zAddressRange(j)$
──

4.1.4 Separation Kernel Boot-time Initialization

At boot time, the HiRTOS separation kernel is initialized when the
`HiRTOS.Separation_Kernel.Initialize` *HiRTOS* API is called on every CPU core:

─── *HiRtosSeparationKernelInitialize* ──────────────
HiRtosSeparationKernel′
SeparationKernelCpuInstanceInitialize
cpuId? : *CpuIdType*
─────────────
cpuId′ = *cpuId?*

$\theta SeparationKernelCpuInstanceType′ = separationKernelCpuInstances′(cpuId?)$
──

─── *SeparationKernelCpuInstanceInitialize* ──────────────
SeparationKernelCpuInstanceElaboration
──

$\quad$*SeparationKernelCpuInstanceElaboration*
$\quad$*SeparationKernelCpuInstanceType$'$*

$zAddressRange(interruptStack') \neq \emptyset$

$partitionSchedulerState' = partitionSchedulerStopped$

$currentPartitionId' = invalidPartitionId$

$timerTicksSinceBoot' = 0$

$runnablePartitionsQueue' = \langle\rangle$

$partitions' = \emptyset$

4.2 Separation Kernel Callable Services

4.2.1 Partition Operations

Create a new partition

A partition can be created by calling `HiRTOS.Separation_Kernel.Partition.Create_Partition`. Partitions are allocated from the pool of partition objects of the calling CPU:

$\quad$*CreatePartition*
$\quad\Delta SeparationKernelCpuInstanceType$
$\quad InitializeNewPartition$

$partitionId! \notin \operatorname{dom} partitions$

$partitionId! \in \operatorname{dom} partitions'$

$partitions'(partitionId!) = partitions'(partitionId!)$

$runnablePartitionsQueue' = runnablePartitionsQueue ^\frown \langle partitionId!\rangle$

InitializeNewPartition ────────────────────────────

$PartitionType'$
$partitionId! : ValidPartitionIdType$

$id' = partitionId!$

$failoverPartitionId' = invalidPartitionId$

$state' = partitionRunnable$

$savedHypervisorLevelMpuRegions' \neq \emptyset$

$savedSupervisorLevelMpuRegions' = \emptyset$

$savedInterruptsEnabled' = \emptyset$

───

Bibliography

[1] Mike Spivey, "Z Reference Card", 1992
`https://github.com/Spivoxity/fuzz/blob/master/doc/refcard3-pub.pdf`

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

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

[4] Mike Spivey, "The Fuzz checker"
`https://github.com/Spivoxity/fuzz`

[5] John W. McCormick, Peter C. Chapin,"Building High Integrity Applications with SPARK", Cambridge University Press, 2015
`https://www.amazon.com/Building-High-Integrity-Applications-SPARK/dp/1107040736`

[6] AdaCore,"Formal Verification with GNATprove"
`https://docs.adacore.com/spark2014-docs/html/ug/en/gnatprove.html`

[7] Andrew D. Birrel, "An Introduction to Programming with Threads", Digital Equipment Corporation, Systems Research Center, 1989
`http://birrell.org/andrew/papers/035-Threads.pdf`

[8] Andrew D. Birrel et al, "Synchronization Primitives for a Multiprocessor: A Formal Specification", Digital Equipment Corporation, Systems Research Center, 1987
`https://dl.acm.org/doi/pdf/10.1145/37499.37509`

[9] ISO, "N2731: Working draft of the C23 standard, section 7.26", October 2021
`http://www.open-std.org/jtc1/sc22/wg14/www/docs/n2596.pdf#page=345&zoom=100,102,113`

[10] Lui Sha et al, "Priority Inheritance Protocols: An Approach to Real-Time Synchronization", IEEE Transactions on Computers, September 1990
`https://www.csie.ntu.edu.tw/~r95093/papers/Priority%20Inheritance%20Protocols%20An%20Approach%20to%20Real-Time%20Synchronization.pdf`

[11] John Rushby, "Design and Verification of Secure Systems", ACM SIGOPS Operating Systems Review, 1981
https://www.csl.sri.com/users/rushby/papers/sosp81.pdf

[12] John Rushby, "Partitioning in Avionics Architectures: Requirements, Mechanisms, and Assurance", Technical Report. 1999
https://www.tc.faa.gov/its/worldpac/techrpt/ar99-58.pdf

[13] FreeRTOS
https://www.freertos.org/

[14] CMSIS-RTOS API v2 (CMSIS-RTOS2)
https://www.keil.com/pack/doc/CMSIS/RTOS2/html/group__CMSIS__RTOS.html

[15] Leslie Lamport, "Specifying Systems: The TLA+ Language and Tools for Hardware and Software Engineers", Addison Wesley, 2003 https://lamport.azurewebsites.net/tla/book.html?back-link=learning.html#book

[16] 'Hillel Wayne, ''Practical TLA+: Planning Driven Development", APress, 2018
https://link.springer.com/book/10.1007/978-1-4842-3829-5

[17] Leslie Lamport, "The TLA+ Toolbox" https://lamport.azurewebsites.net/tla/toolbox.html