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

Germán Rivera
jgrivera67@gmail.com

April 18, 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 assumptions 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 as such are 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 assumptions as part of the code. One challenge when doing programming-by-contract is to be aware of all the correctness assumptions that can be checked in programming contracts. Describing software design in a formal notation, such as the Z notation [1, 2, 3], can help identify/elicit correctness assumptions 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. HiRTOS data types were modeled in Z at a level of abstraction that can be mapped directly to corresponding data types in SPARK Ada.

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.
- *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 between CPU cores is supported by *HiRTOS*, so that the *HiRTOS* API can stay the same for both single-core and multi-core platforms. Inter-core communication 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 application threads run in unprivileged mode. For each thread, the only writable memory, by default, is its own stack and global variables. Stacks of other threads are not accessible. MMIO space is only accessible to privileged code, by default. Application driver code, other than ISRs, must request access (read-only or read-write permission) to *HiRTOS* via a system call.

- Interrupt service routines (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].

- 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, so that the separation kernel API can stay the same for both single-core and multi-core platforms. Inter-core communication would need to be provided outside of HiRTOS, 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 more disjoint address ranges covering portions of RAM and MMIO space that only that partition can access. Also, each partition has its own interrupt vector table, its own set of physical interrupts and its own global machine state. So, the firmware hosted in each partition has the illusion that it owns an entire physical machine, with is own set of of physical peripherals, dedicated memory and CPU core. 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 [12] and RTOS interfaces such as the CMSIS RTOS2 API [13]. 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 there is a bare-metal Ada runtime library available with the GNAT Ada compiler. an all platforms where is available now or in the future.

HiRTOS has been architected to be easily portable to any 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.

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.

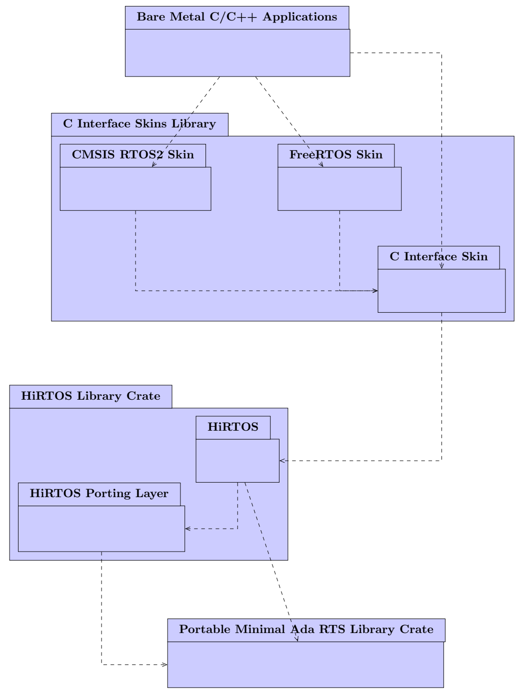

Figure 2.1: HiRTOS Code Architecture for C/C++ Applications

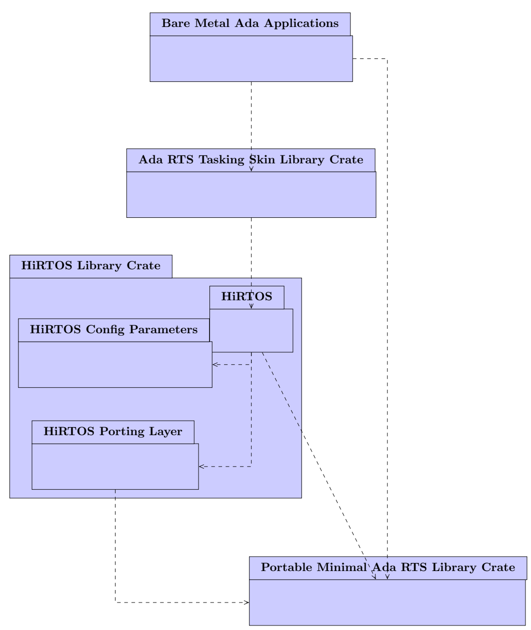

Figure 2.2: HiRTOS Code Architecture for Ada Applications

The *HiRTOS porting layer* contains hardware-dependent components that provide hardware-independent interfaces to upper HiRTOS layers.

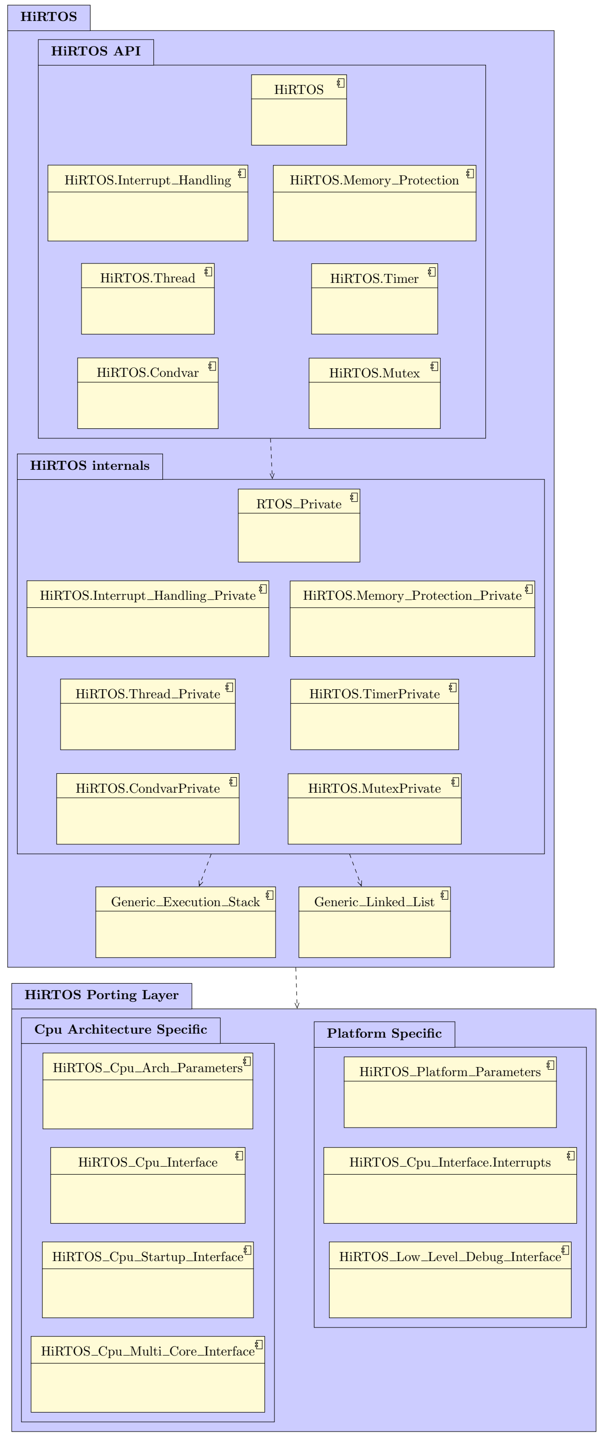

Figure 2.3: HiRTOS Code Components

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*.

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

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 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*.

$$
\begin{array}{|l}
numCpus : \mathbb{N}_1 \\
minMemoryAddress : \mathbb{N} \\
maxMemoryAddress : \mathbb{N}_1 \\
numInterruptPriorities : \mathbb{N}_1 \\
maxNumInterrupts : \mathbb{N}_1 \\
\hline
minMemoryAddress < maxMemoryAddress
\end{array}
$$

3.1.4 *HiRTOS* Primitive Types

Below are the primitive types used in HiRTOS:

$$
\begin{array}{l}
[CpuIdType] \\
\#CpuIdType = numCpus + 1 \\
[ThreadIdType] \\
\#ThreadIdType = maxNumThreads + 1 \\
[MutexIdType] \\
\#MutexIdType = maxNumMutexes + 1 \\
[CondvarIdType] \\
\#CondvarIdType = maxNumCondvars + 1 \\
[TimerIdType] \\
\#TimerIdType = maxNumTimers + 1 \\
[InterruptIdType] \\
\#InterruptIdType = maxNumInterrupts + 1
\end{array}
$$

$$
\begin{array}{|l}
invalidCpuId : CpuIdType \\
invalidThreadId : ThreadIdType \\
invalidMutexId : MutexIdType \\
invalidCondvarId : CondvarIdType \\
invalidTimerId : TimerIdType \\
invalidInterruptId : InterruptIdType
\end{array}
$$

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

$$
\begin{aligned}
&\mathit{ThreadQueueType} == \operatorname{iseq} \mathit{ValidThreadIdType}\\
&\mathit{MutexListType} == \operatorname{iseq} \mathit{ValidMutexIdType}\\
&\mathit{TimerWheelSpokeIndexType} ==\\
&\qquad 0 \mathrel{..} \mathit{numTimerWheelSpokes}\\
&\mathit{invalidTimerWheelSpokeIndex} ==\\
&\qquad \mathit{max}\ \mathit{TimerWheelSpokeIndexType}\\
&\mathit{ValidTimerWheelSpokeIndexType} ==\\
&\qquad \mathit{TimerWheelSpokeIndexType} \setminus\\
&\qquad \{\ \mathit{invalidTimerWheelSpokeIndex}\ \}\\
&\mathit{TimerKindType} ::= \mathit{periodicTimer} \mid \mathit{oneShotTimer}\\
&\mathit{TimerStateType} ::= \mathit{timerStopped} \mid \mathit{timerRunning}
\end{aligned}
$$

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

3.1.5 *HiRTOS* Axiomatic Definitions

$$
\begin{array}{|l}
\mathit{zAddressRange} :\\
\qquad (\mathit{MemoryAddressType} \times \mathit{MemoryAddressType}) \rightarrowtail\\
\qquad \mathbb{F}_1\ \mathit{MemoryAddressType}\\
\hline
\forall\, x, y : \mathit{MemoryAddressType} \mid\\
\qquad (x, y) \in \operatorname{dom} \mathit{zAddressRange} \bullet\\
\qquad x < y \land \mathit{zAddressRange}(x, y) = x \mathrel{..} y
\end{array}
$$

$$
\begin{array}{|l}
\mathit{zCpuToISRstackAddressRange} :\\
\qquad \mathit{ValidCpuIdType} \rightarrowtail\\
\qquad (\mathit{MemoryAddressType} \times \mathit{MemoryAddressType})\\
\hline
\bigcap \{\ i : \mathit{ValidCpuIdType} \bullet\\
\mathit{zAddressRange}(\mathit{zCpuToISRstackAddressRange}(i))\ \} = \emptyset\\
\\
\forall\, i : \operatorname{dom} \mathit{zCpuToISRstackAddressRange} \bullet\\
\#(\mathit{zAddressRange}(\mathit{zCpuToISRstackAddressRange}(i))) \geq 2
\end{array}
$$

$$
\begin{array}{|l}
\mathit{interruptPriorities} :\\
\qquad \mathit{InterruptIdType} \rightarrow \mathit{InterruptPriorityType}
\end{array}
$$

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

$$
\begin{array}{l}
rtosCpuInstances : \\
\quad ValidCpuIdType \twoheadrightarrow\!\!\!\!\!\!\rightarrow HiRtosCpuInstanceType \\
zAllCreatedThreadInstances : \mathbb{F}\ ThreadType \\
\hline
\#rtosCpuInstances \geq 1 \\[1ex]
\forall i : \operatorname{dom} rtosCpuInstances \bullet \\
\quad (rtosCpuInstances(i)).cpuId = i \\[1ex]
zAllCreatedThreadInstances = \\
\quad \bigcup \{ i : ValidCpuIdType \bullet \\
\quad\quad \operatorname{ran}(rtosCpuInstances(i)).threads \} \\[1ex]
\bigcap \{ i : ValidCpuIdType \bullet \\
\quad \operatorname{ran}(rtosCpuInstances(i)).threads \} = \emptyset \\[1ex]
\bigcap \{ thread : zAllCreatedThreadInstances \bullet \\
\quad zAddressRange(thread.stack) \} = \emptyset \\[1ex]
\bigcap \{ i : ValidCpuIdType \bullet \\
\quad \operatorname{ran}(rtosCpuInstances(i)).mutexes \} = \emptyset \\[1ex]
\bigcap \{ i : ValidCpuIdType \bullet \\
\quad \operatorname{ran}(rtosCpuInstances(i)).condvars \} = \emptyset \\[1ex]
\bigcap \{ i : ValidCpuIdType \bullet \\
\quad \operatorname{ran}(rtosCpuInstances(i)).timers \} = \emptyset \\[1ex]
\bigcap \{ t : zAllCreatedThreadInstances \bullet zAddressRange(t.stack) \} = \emptyset
\end{array}
$$

Per-CPU HiRTOS Instance

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

$$
\begin{array}{|l}
\hline
\textit{HiRtosCpuInstanceType} \\
\textit{cpuId} : \textit{CpuIdType} \\
\textit{currentCpuContext} : \textit{CpuRegistersType} \\
\textit{threadSchedulerState} : \textit{ThreadSchedulerStateType} \\
\textit{currentAtomicLevel} : \textit{AtomicLevelType} \\
\textit{currentCpuExecutionMode} : \textit{CpuExecutionModeType} \\
\textit{currentThreadId} : \textit{ThreadIdType} \\
\textit{timerTicksSinceBoot} : \mathbb{N} \\
\textit{idleThreadId} : \textit{ValidThreadIdType} \\
\textit{tickTimerThreadId} : \textit{ValidThreadIdType} \\
\textit{interruptNestingLevelStack} : \textit{InterruptNestingLevelStackType} \\
\textit{threads} : \textit{ValidThreadIdType} \rightarrowtail\!\!\!\!\!\!\!\!\twoheadrightarrow\!\!\!\!\!\!\!+ \textit{ThreadType} \\
\textit{mutexes} : \textit{ValidMutexIdType} \rightarrowtail\!\!\!\!\!\!\!\!\twoheadrightarrow\!\!\!\!\!\!\!+ \textit{MutexType} \\
\textit{condvars} : \textit{ValidCondvarIdType} \rightarrowtail\!\!\!\!\!\!\!\!\twoheadrightarrow\!\!\!\!\!\!\!+ \textit{CondvarType} \\
\textit{timers} : \textit{ValidTimerIdType} \rightarrowtail\!\!\!\!\!\!\!\!\twoheadrightarrow\!\!\!\!\!\!\!+ \textit{TimerType} \\
\textit{runnableThreadsQueue} : \textit{ThreadPriorityQueueType} \\
\textit{timerWheel} : \textit{TimerWheelType} \\
\textit{zCpuInterruptMaskingState} : \textit{CpuInterruptMaskingStateType} \\
\textit{zCpuPrivilege} : \textit{CpuPrivilegeType} \\
\textit{zMemoryProtectionState} : \textit{MemoryProtectionStateType} \\
\hline
\{\, \textit{idleThreadId}, \textit{tickTimerThreadId} \,\} \subseteq \operatorname{dom} \textit{threads} \\
\textit{tickTimerThreadId} \neq \textit{idleThreadId} \\
\textit{threadSchedulerState} = \textit{threadSchedulerRunning} \Rightarrow \\
(\forall t : \operatorname{ran}(\{\, \textit{currentThreadId} \,\} \lhd \textit{threads}) \bullet \\
\quad t.\textit{currentPriority} < (\textit{threads}(\textit{currentThreadId})).\textit{currentPriority}) \\
\textit{zCpuInterruptMaskingState} = \textit{cpuInterruptsEnabled} \Leftrightarrow \\
\quad \textit{currentAtomicLevel} > \textit{atomicLevelNoInterrupts} \\
\textit{zCpuInterruptMaskingState} = \textit{cpuInterruptsDisabled} \Rightarrow \textit{zCpuPrivilege} = \textit{cpuPrivileged} \\
\textit{currentAtomicLevel} < \textit{atomicLevelNone} \Rightarrow \textit{zCpuPrivilege} = \textit{cpuPrivileged} \\
\bigcap \{\, t : \operatorname{ran} \textit{threads} \bullet \{\, t.\textit{builtinCondvarId} \,\} \,\} = \emptyset \\
\bigcap \{\, t : \operatorname{ran} \textit{threads} \bullet \{\, t.\textit{builtinTimerId} \,\} \,\} = \emptyset \\
\forall t : \operatorname{ran} \textit{threads} \bullet (t.\textit{id} = (\textit{threads}^{\sim})(t) \land \\
(t.\textit{ownedMutexes} \neq \emptyset \Rightarrow \\
t.\textit{currentPriority} = \textit{max} \{\, m : \operatorname{ran} t.\textit{ownedMutexes} \bullet (\textit{mutexes}(m)).\textit{ceilingPriority} \,\})) \\
\forall m : \operatorname{ran} \textit{mutexes} \bullet m.\textit{id} = \textit{mutexes}^{\sim}(m) \\
\forall c : \operatorname{ran} \textit{condvars} \bullet c.\textit{id} = \textit{condvars}^{\sim}(c) \\
\forall \textit{ti} : \operatorname{ran} \textit{timers} \bullet \textit{ti}.\textit{id} = \textit{timers}^{\sim}(\textit{ti}) \\
\forall p : \textit{ValidThreadPriorityType} \bullet \\
\quad \forall \textit{threadId} : \operatorname{ran}(\textit{runnableThreadsQueue}.\textit{threadQueues}(p)) \bullet \\
\qquad (\textit{threads}(\textit{threadId})).\textit{currentPriority} = p \\
\hline
\end{array}
$$

ThreadPriorityQueueType

$$
\begin{array}{l}
\mathit{threadQueues} : \\
\quad \mathit{ValidThreadPriorityType} \rightarrowtail \mathit{ThreadQueueType} \\
\mathit{waitingThreadsCount} : \mathbb{N} \\
\hline
\mathit{waitingThreadsCount} = \\
\#(\bigcup \{\, p : \mathit{ValidThreadPriorityType} \bullet \mathit{threadQueues}(p) \,\})
\end{array}
$$

InterruptNestingLevelStackType

$$
\begin{array}{l}
\mathit{interruptNestingLevels} : \\
\quad \mathit{ActiveInterruptNestingCounterType} \rightarrowtail \\
\quad \mathit{InterruptNestingLevelType} \\
\mathit{currentInterruptNestingCounter} : \\
\quad \mathit{InterruptNestingCounterType} \\
\mathit{zCpuId} : \mathit{ValidCpuIdType} \\
\hline
\forall x : \mathrm{dom}\, \mathit{interruptNestingLevels} \bullet \\
\quad (\mathit{interruptNestingLevels}(x)).\mathit{interruptNestingCounter} = x \\
\quad \land \\
\quad (\mathit{interruptNestingLevels}(x)).\mathit{savedStackPointer} \in \\
\quad \mathit{zAddressRange}(\mathit{zCpuToISRstackAddressRange}(\mathit{zCpuId})) \\
\\
\mathrm{dom}\, \mathit{interruptNestingLevels} = \\
1 \mathrel{..} \mathit{currentInterruptNestingCounter}
\end{array}
$$

InterruptNestingLevelType

$$
\begin{array}{l}
\mathit{interruptId} : \mathit{InterruptIdType} \\
\mathit{interruptNestingCounter} : \mathit{ActiveInterruptNestingCounterType} \\
\mathit{savedStackPointer} : \mathit{MemoryAddressType} \\
\mathit{atomicLevel} : \mathit{AtomicLevelType} \\
\hline
\mathit{atomicLevel} \leq \mathit{interruptPriorities}(\mathit{interruptId})
\end{array}
$$

TimerWheelType

$$
\begin{array}{l}
\mathit{wheelSpokesHashTable} : \\
\quad \mathit{ValidTimerWheelSpokeIndexType} \rightarrow \mathbb{F}\, \mathit{ValidTimerIdType} \\
\mathit{currentWheelSpokeIndex} : \mathit{ValidTimerWheelSpokeIndexType} \\
\hline
\forall i, j : \mathit{ValidTimerWheelSpokeIndexType} \mid i \neq j \bullet \\
\quad \mathit{wheelSpokesHashTable}(i) \cap \mathit{wheelSpokesHashTable}(j) = \emptyset
\end{array}
$$

TimerType

$$
\begin{array}{l}
id : TimerIdType \\
timerKind : TimerKindType \\
timerState : TimerStateType \\
timerWheelRevolutions : \mathbb{N} \\
timerWheelRevolutionsLeft : \mathbb{N} \\
expirationCallbackAddr : MemoryAddressType \\
wheelSpokeIndex : TimerWheelSpokeIndexType \\
\hline
timerWheelRevolutionsLeft \leq timerWheelRevolutions \\
timerState = timerRunning \Rightarrow expirationCallbackAddr \neq nullAddress
\end{array}
$$

$$
\begin{array}{|l}
\textit{ThreadType} \\
\hline
id : ThreadIdType \\
state : ThreadStateType \\
currentPriority : ThreadPriorityType \\
basePriority : ThreadPriorityType \\
atomicLevel : AtomicLevelType \\
builtinTimerId : TimerIdType \\
builtinCondvarId : CondvarIdType \\
waitingOnCondvarId : CondvarIdType \\
waitingOnMutexId : MutexIdType \\
ownedMutexes : \mathrm{iseq}\ ValidMutexIdType \\
savedStackPointer : MemoryAddressType \\
stack : AddressRangeType \\
stackSavedCpuContext : CpuRegistersType \\
privilegedNestingCounter : \mathbb{N} \\
timeSliceLeftUs : \mathbb{N} \\
\hline
state \neq threadNotCreated \Rightarrow \\
\quad (id \neq invalidThreadId \land \\
\quad builtinTimerId \neq invalidTimerId \land \\
\quad builtinCondvarId \neq invalidCondvarId \land \\
\quad basePriority \neq invalidThreadPriority \land \\
\quad currentPriority \neq invalidThreadPriority \land \\
\quad savedStackPointer \in zAddressRange(stack)) \\
currentPriority \geq basePriority \\
state = threadBlockedOnCondvar \Leftrightarrow \\
\quad waitingOnCondvarId \neq invalidCondvarId \\
state = threadBlockedOnMutex \Leftrightarrow \\
\quad waitingOnMutexId \neq invalidMutexId \\
waitingOnCondvarId \neq invalidCondvarId \Rightarrow \\
\quad waitingOnMutexId = invalidMutexId \\
waitingOnMutexId \neq invalidMutexId \Rightarrow \\
\quad waitingOnCondvarId = invalidCondvarId \\
waitingOnMutexId \notin \mathrm{ran}\ ownedMutexes \\
\hline
\end{array}
$$

$$
\begin{array}{l}
\textbf{CondvarType} \\
\hline
id : CondvarIdType \\
wakeupAtomicLevel : AtomicLevelType \\
wakeupMutexId : MutexIdType \\
waitingThreadsQueue : ThreadPriorityQueueType \\
\hline
wakeupAtomicLevel \neq atomicLevelNone \Rightarrow wakeupMutexId = invalidMutexId
\end{array}
$$

$$
\begin{array}{l}
\textbf{MutexType} \\
\hline
id : MutexIdType \\
ownerThreadId : ThreadIdType \\
recursiveCount : \mathbb{N} \\
ceilingPriority : ThreadPriorityType \\
waitingThreadsQueue : ThreadPriorityQueueType \\
\hline
waitingThreadsQueue.waitingThreadsCount \neq 0 \Rightarrow \\
\quad ownerThreadId \neq invalidThreadId \\
ceilingPriority \neq invalidThreadPriority \Rightarrow \\
(\forall\, p : ValidThreadPriorityType \mid \\
\quad waitingThreadsQueue.threadQueues(p) \neq \emptyset \bullet \\
\quad p \leq ceilingPriority)
\end{array}
$$

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:

$$
\begin{array}{l}
\textbf{HiRtosInitialize} \\
\hline
HiRtos' \\
HiRtosCpuInstanceInitialize \\
cpuId? : CpuIdType \\
\hline
cpuId' = cpuId? \\
\theta HiRtosCpuInstanceType' = rtosCpuInstances'(cpuId?) \\
interruptNestingLevelStack'.zCpuId = cpuId?
\end{array}
$$

HiRtosCpuInstanceInitialize

HiRtosCpuInstanceElaboration

$idleThreadId' \neq invalidThreadId$

$tickTimerThreadId' \neq invalidThreadId$

$tickTimerThreadId' \neq idleThreadId'$

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

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

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

$zCpuInterruptMaskingState' = cpuInterruptsEnabled$

$zCpuPrivilege' = cpuUnprivileged$

$zMemoryProtectionState' = memoryProtectionOn$

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

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

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

$$
\begin{array}{|l}
\hline
\textit{HiRtosCpuInstanceElaboration} \\
\textit{HiRtosCpuInstanceType}' \\
\textit{InterruptNestingLevelStackElaboration} \\
\hline
\textit{threadSchedulerState}' = \textit{threadSchedulerStopped} \\
\textit{currentThreadId}' = \textit{invalidThreadId} \\
\textit{currentAtomicLevel}' = \textit{atomicLevelNone} \\
\textit{currentCpuExecutionMode}' = \textit{cpuExecutingResetHandler} \\
\textit{idleThreadId}' = \textit{invalidThreadId} \\
\textit{tickTimerThreadId}' = \textit{invalidThreadId} \\
\textit{threads}' = \emptyset \\
\textit{condvars}' = \emptyset \\
\textit{timers}' = \emptyset \\
\textit{timerTicksSinceBoot}' = 0 \\
\hline
\end{array}
$$

$$
\begin{array}{|l}
\hline
\textit{InterruptNestingLevelStackElaboration} \\
\textit{InterruptNestingLevelStackType}' \\
\textit{InterruptNestingLevelElaboration} \\
\hline
\textit{currentInterruptNestingCounter}' = 1 \\
\forall x : \textit{ActiveInterruptNestingCounterType} \bullet \\
\qquad \textit{interruptNestingLevels}'(x) = \theta \textit{InterruptNestingLevelType}' \\
\hline
\end{array}
$$

$$
\begin{array}{|l}
\hline
\textit{InterruptNestingLevelElaboration} \\
\textit{InterruptNestingLevelType}' \\
\hline
\textit{interruptId}' = \textit{invalidInterruptId} \\
\textit{interruptNestingCounter}' = 0 \\
\textit{savedStackPointer}' = \textit{nullAddress} \\
\textit{atomicLevel}' = \textit{atomicLevelNone} \\
\hline
\end{array}
$$

$$
\begin{array}{l}
\textit{TimerWheelElaboration} \\
\hline
\textit{TimerWheelType}' \\
\hline
\mathrm{ran}\ \textit{wheelSpokesHashTable}' = \{ \emptyset \} \\
\textit{currentWheelSpokeIndex}' = \\
\quad \textit{min}\ \textit{ValidTimerWheelSpokeIndexType}
\end{array}
$$

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:

$$
\begin{array}{l}
\textit{CreateThread} \\
\hline
\Delta \textit{HiRtosCpuInstanceType} \\
\textit{InitializeNewThread} \\
\textit{priority}? : \textit{ValidThreadPriorityType} \\
\hline
\textit{threadId}! \notin \mathrm{dom}\ \textit{threads} \\
\textit{condvarId}! \notin \mathrm{dom}\ \textit{condvars} \\
\textit{timerId}! \notin \mathrm{dom}\ \textit{timers} \\
\textit{threadId}! \in \mathrm{dom}\ \textit{threads}' \\
\textit{condvarId}! \in \mathrm{dom}\ \textit{condvars}' \\
\textit{timerId}! \in \mathrm{dom}\ \textit{timers}' \\
\textit{threads}'(\textit{threadId}!) = \theta\, \textit{ThreadType}' \\
\textit{runnableThreadsQueue}'.\textit{threadQueues}(\textit{priority}?) = \\
\quad \textit{runnableThreadsQueue}.\textit{threadQueues}(\textit{priority}?) \frown \langle \textit{threadId}! \rangle
\end{array}
$$

$$
\begin{array}{|l}
\hline
\textit{InitializeNewThread} \\
\textit{ThreadType}' \\
\textit{condvarId}! : \textit{ValidCondvarIdType} \\
\textit{timerId}! : \textit{ValidTimerIdType} \\
\textit{threadId}! : \textit{ValidThreadIdType} \\
\hline
\textit{threadId}! \neq \textit{invalidThreadId} \\
\textit{id}' = \textit{threadId}! \\
\textit{builtinTimerId}' = \textit{timerId}! \\
\textit{builtinCondvarId}' = \textit{condvarId}! \\
\textit{state}' = \textit{threadRunnable} \\
\textit{atomicLevel}' = \textit{atomicLevelNone} \\
\hline
\end{array}
$$

$$
\begin{array}{|l}
\hline
\textit{ThreadPriorityQueueInitialize} \\
\textit{ThreadPriorityQueueType}' \\
\hline
\forall\, p : \textit{ValidThreadPriorityType} \bullet \\
\qquad \textit{threadQueues}'(p) = \emptyset \\
\hline
\end{array}
$$

$$
\begin{array}{|l}
\hline
\textit{DequeueHighestPriorityThread} \\
\Delta \textit{ThreadPriorityQueueType} \\
\textit{highestPriorityThreadId}! : \textit{ValidThreadIdType} \\
\hline
(\mathbf{let}\ \textit{highestPrio} == \\
\qquad \textit{max}\,(\mathrm{dom}(\textit{threadQueues} \rhd \emptyset)) \bullet \\
\qquad \textit{highestPriorityThreadId}! = \textit{head}\,(\textit{threadQueues}(\textit{highestPrio})) \land \\
\qquad \textit{threadQueues}'(\textit{highestPrio}) = \textit{tail}\,(\textit{threadQueues}(\textit{highestPrio}))) \\
\hline
\end{array}
$$

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:

$$
\begin{array}{|l}
\hline
\textit{CreateMutex} \\
\Delta \textit{HiRtosCpuInstanceType} \\
\textit{InitializeNewMutex} \\
\hline
\textit{mutexId}! \notin \operatorname{dom} \textit{mutexes} \\
\textit{mutexId}! \in \operatorname{dom} \textit{mutexes}' \\
\theta \textit{MutexType}' = \textit{mutexes}'(\textit{mutexId}!) \\
\hline
\end{array}
$$

$$
\begin{array}{|l}
\hline
\textit{InitializeNewMutex} \\
\textit{MutexType}' \\
\textit{ceilingPriority}? : \textit{ThreadPriorityType} \\
\textit{mutexId}! : \textit{ValidMutexIdType} \\
\hline
\textit{mutexId}! \neq \textit{invalidMutexId} \\
\textit{id}' = \textit{mutexId}! \\
\textit{ownerThreadId}' = \textit{invalidThreadId} \\
\textit{recursiveCount}' = 0 \\
\textit{ceilingPriority}' = \textit{ceilingPriority}? \\
\forall p : \textit{ValidThreadPriorityType} \bullet \\
\quad \textit{waitingThreadsQueue}'.\textit{threadQueues}(p) = \emptyset \\
\hline
\end{array}
$$

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`:

$$
\begin{array}{|l}
\hline
\textit{CreatedMutexMutableOperation} \\
\Delta HiRtos \\
\Delta HiRtosCpuInstanceType \\
\Delta MutexType \\
cpuId? : CpuIdType \\
mutexId? : ValidMutexIdType \\
\hline
\theta HiRtosCpuInstanceType = rtosCpuInstances(cpuId?) \\
\theta HiRtosCpuInstanceType' = rtosCpuInstances'(cpuId?) \\
\theta MutexType = mutexes(mutexId?) \\
\theta MutexType' = mutexes'(mutexId?) \\
\hline
\end{array}
$$

Acquire a mutex

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

$$
\begin{array}{|l}
\hline
\textit{AcquireAvailableMutex} \\
CreatedMutexMutableOperation \\
\hline
ownerThreadId = invalidThreadId \lor \\
(ownerThreadId = currentThreadId \Rightarrow \\
recursiveCount' = recursiveCount + 1) \\
ownerThreadId' = currentThreadId \\
(ceilingPriority \neq invalidThreadPriority \land \\
(threads(currentThreadId)).currentPriority < \\
\quad ceilingPriority) \Rightarrow \\
(threads'(currentThreadId)).currentPriority = \\
\quad ceilingPriority \\
(threads'(currentThreadId)).ownedMutexes = \\
\quad (threads(currentThreadId)).ownedMutexes \frown \langle mutexId? \rangle \\
\hline
\end{array}
$$

WaitOnUnavailableMutex

$CreatedMutexMutableOperation$
$DequeueHighestPriorityThread$

$runnableThreadsQueue = \theta\, ThreadPriorityQueueType$

$runnableThreadsQueue' = \theta\, ThreadPriorityQueueType'$

$ownerThreadId \neq invalidThreadId$

$currentThreadId \neq ownerThreadId$

$currentThreadId' \neq currentThreadId$

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

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

$currentThreadId' \neq currentThreadId$

$currentThreadId' = highestPriorityThreadId!$

$$
\begin{array}{l}
AcquireMutex \mathrel{\widehat{=}} \\
\quad AcquireAvailableMutex \lor WaitOnUnavailableMutex
\end{array}
$$

Release a mutex

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

ReleaseMutex

$$\mathit{CreatedMutexMutableOperation}$$
$$\mathit{DequeueHighestPriorityThread}$$

$$\mathit{waitingThreadsQueue} = \theta\,\mathit{ThreadPriorityQueueType}$$

$$\mathit{waitingThreadsQueue}' = \theta\,\mathit{ThreadPriorityQueueType}'$$

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

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

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

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

$$\Delta \mathit{HiRtosCpuInstanceType}$$
$$\mathit{InitializeNewCondvar}$$

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

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

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

$$
\begin{array}{|l}
\hline
\textit{InitializeNewCondvar} \\
\textit{CondvarType}' \\
\textit{ThreadPriorityQueueInitialize} \\
\textit{condvarId}! : \textit{ValidCondvarIdType} \\
\hline
\textit{waitingThreadsQueue}' = \theta\,\textit{ThreadPriorityQueueType}' \\
\textit{condvarId}! \neq \textit{invalidCondvarId} \\
\textit{id}' = \textit{condvarId}! \\
\textit{wakeupAtomicLevel}' = \textit{atomicLevelNone} \\
\textit{wakeupMutexId}' = \textit{invalidMutexId} \\
\hline
\end{array}
$$

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`:

$$
\begin{array}{|l}
\hline
\textit{CreatedCondvarMutableOperation} \\
\Delta \textit{HiRtos} \\
\Delta \textit{HiRtosCpuInstanceType} \\
\Delta \textit{CondvarType} \\
\textit{cpuId}? : \textit{CpuIdType} \\
\textit{condvarId}? : \textit{ValidCondvarIdType} \\
\hline
\textit{rtosCpuInstances}(\textit{cpuId}?) = \theta\,\textit{HiRtosCpuInstanceType} \\
\textit{rtosCpuInstances}'(\textit{cpuId}?) = \theta\,\textit{HiRtosCpuInstanceType}' \\
\textit{condvars}(\textit{condvarId}?) = \theta\,\textit{CondvarType} \\
\textit{condvars}'(\textit{condvarId}?) = \theta\,\textit{CondvarType}' \\
\hline
\end{array}
$$

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:

$$
\begin{array}{|l}
\textit{WaitOnCondvar} \\
\hline
\textit{CreatedCondvarMutableOperation} \\
\textit{DequeueHighestPriorityThread} \\
\textit{mutexId?} : \textit{ValidMutexIdType} \\
\hline
\textit{runnableThreadsQueue} = \theta\, \textit{ThreadPriorityQueueType} \\
\textit{runnableThreadsQueue}' = \theta\, \textit{ThreadPriorityQueueType}' \\
\textit{mutexId?} \in \operatorname{ran}(\textit{threads}(\textit{currentThreadId})).\textit{ownedMutexes} \\
\textit{mutexId?} \notin \operatorname{ran}(\textit{threads}'(\textit{currentThreadId})).\textit{ownedMutexes} \\
\textit{wakeupMutexId}' = \textit{mutexId?} \\
(\mathbf{let}\ p == (\textit{threads}(\textit{currentThreadId})).\textit{currentPriority} \bullet \\
\quad \textit{waitingThreadsQueue}'.\textit{threadQueues}(p) = \\
\quad\quad \textit{waitingThreadsQueue}.\textit{threadQueues}(p) \frown \langle \textit{currentThreadId} \rangle) \\
\textit{currentThreadId}' \neq \textit{currentThreadId} \\
\textit{currentThreadId}' = \textit{highestPriorityThreadId!} \\
\hline
\end{array}
$$

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.

$$
\begin{array}{|l}
\textit{SignalCondvar} \\
\hline
\textit{CreatedCondvarMutableOperation} \\
\textit{DequeueHighestPriorityThread} \\
\hline
\textit{waitingThreadsQueue} = \theta\, \textit{ThreadPriorityQueueType} \\
\textit{waitingThreadsQueue}' = \theta\, \textit{ThreadPriorityQueueType}' \\
\textit{wakeupMutexId} \in \operatorname{ran}(\textit{threads}'(\textit{highestPriorityThreadId!})).\textit{ownedMutexes} \\
(\mathbf{let}\ p == (\textit{threads}(\textit{highestPriorityThreadId!})).\textit{currentPriority} \bullet \\
\quad \textit{runnableThreadsQueue}'.\textit{threadQueues}(p) = \\
\quad\quad \textit{runnableThreadsQueue}.\textit{threadQueues}(p) \frown \langle \textit{highestPriorityThreadId!} \rangle) \\
\hline
\end{array}
$$

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

$$
\begin{array}{l}
\mathit{CreatedCondvarMutableOperation} \\
\hline
(\forall\, p : \mathit{ValidThreadPriorityType} \mid \\
\quad \mathit{waitingThreadsQueue}.\mathit{threadQueues}(p) \neq \emptyset \bullet \\
\quad \mathit{waitingThreadsQueue}'.\mathit{threadQueues}(p) = \emptyset \land \\
\quad \mathit{runnableThreadsQueue}'.\mathit{threadQueues}(p) = \\
\qquad \mathit{runnableThreadsQueue}.\mathit{threadQueues}(p) ^\frown \\
\qquad \mathit{waitingThreadsQueue}.\mathit{threadQueues}(p))
\end{array}
$$

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

$$
\begin{array}{l}
\Delta \mathit{HiRtosCpuInstanceType} \\
\mathit{InitializeNewTimer} \\
\hline
\mathit{timerId}! \notin \mathrm{dom}\ \mathit{timers} \\
\mathit{timerId}! \in \mathrm{dom}\ \mathit{timers}' \\
\mathit{timers}'(\mathit{timerId}!) = \theta\, \mathit{TimerType}'
\end{array}
$$

InitializeNewTimer

$$
\begin{array}{l}
\mathit{TimerType}' \\
\mathit{timerId}! : \mathit{ValidTimerIdType} \\
\hline
\mathit{timerId}! \neq \mathit{invalidTimerId} \\
\mathit{id}' = \mathit{timerId}! \\
\mathit{timerKind}' = \mathit{oneShotTimer} \\
\mathit{timerState}' = \mathit{timerStopped} \\
\mathit{timerWheelRevolutions}' = 0 \\
\mathit{timerWheelRevolutionsLeft}' = 0 \\
\mathit{expirationCallbackAddr}' = \mathit{nullAddress} \\
\mathit{wheelSpokeIndex}' = \mathit{invalidTimerWheelSpokeIndex}
\end{array}
$$

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`:

$$
\begin{array}{l}
\textit{CreatedTimerMutableOperation} \\
\hline
\Delta \textit{HiRtos} \\
\Delta \textit{HiRtosCpuInstanceType} \\
\Delta \textit{TimerType} \\
\textit{cpuId}? : \textit{CpuIdType} \\
\textit{timerId}? : \textit{ValidTimerIdType} \\
\hline
\textit{rtosCpuInstances}(\textit{cpuId}?) = \theta \textit{HiRtosCpuInstanceType} \\
\textit{rtosCpuInstances}'(\textit{cpuId}?) = \theta \textit{HiRtosCpuInstanceType}' \\
\textit{timers}(\textit{timerId}?) = \theta \textit{TimerType} \\
\textit{timers}'(\textit{timerId}?) = \theta \textit{TimerType}' \\
\hline
\end{array}
$$

Start a software timer

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

$$
\begin{array}{l}
\textit{StartTimer} \\
\hline
\textit{CreatedTimerMutableOperation} \\
\textit{expirationTimeUs}? : \mathbb{N} \\
\textit{expirationCallbackAddr}? : \textit{MemoryAddressType} \\
\textit{timerKind}? : \textit{TimerKindType} \\
\hline
\textit{expirationCallbackAddr}? \neq \textit{nullAddress} \\
\textit{expirationCallbackAddr}' = \textit{expirationCallbackAddr}? \\
\textit{timerKind}' = \textit{timerKind}? \\
(\mathbf{let}\ \textit{expirationTimeTicks} == \textit{expirationTimeUs}?\ \mathsf{div}\ \textit{TickTimerPeriodUs} \bullet \\
\quad \textit{wheelSpokeIndex}' = \\
\qquad (\textit{timerWheel}.\textit{currentWheelSpokeIndex} + \textit{expirationTimeTicks})\ \mathsf{mod}\ \textit{numTimerW} \\
\quad \textit{timerWheelRevolutions}' = \textit{expirationTimeTicks}\ \mathsf{div}\ \textit{numTimerWheelSpokes} \land \\
\quad \textit{timerWheelRevolutionsLeft}' = \textit{timerWheelRevolutions}' \land \\
\quad \textit{timerWheel}'.\textit{wheelSpokesHashTable}(\textit{wheelSpokeIndex}') = \\
\qquad \textit{timerWheel}.\textit{wheelSpokesHashTable}(\textit{wheelSpokeIndex}') \cup \{\ \textit{timerId}?\ \}) \\
\hline
\end{array}
$$

Stop a software timer

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

$$
\begin{array}{|l}
\multicolumn{1}{l}{\textit{StopTimer}} \\
\hline
\textit{CreatedTimerMutableOperation} \\
\hline
\textit{expirationCallbackAddr}' = \textit{nullAddress} \\
\\
\textit{timerWheel}'.\textit{wheelSpokesHashTable}(\textit{wheelSpokeIndex}) = \\
\quad \textit{timerWheel}.\textit{wheelSpokesHashTable}(\textit{wheelSpokeIndex}) \setminus \{ \textit{timerId}? \} \\
\hline
\end{array}
$$

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 threadś CPU context is saved on threadś 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

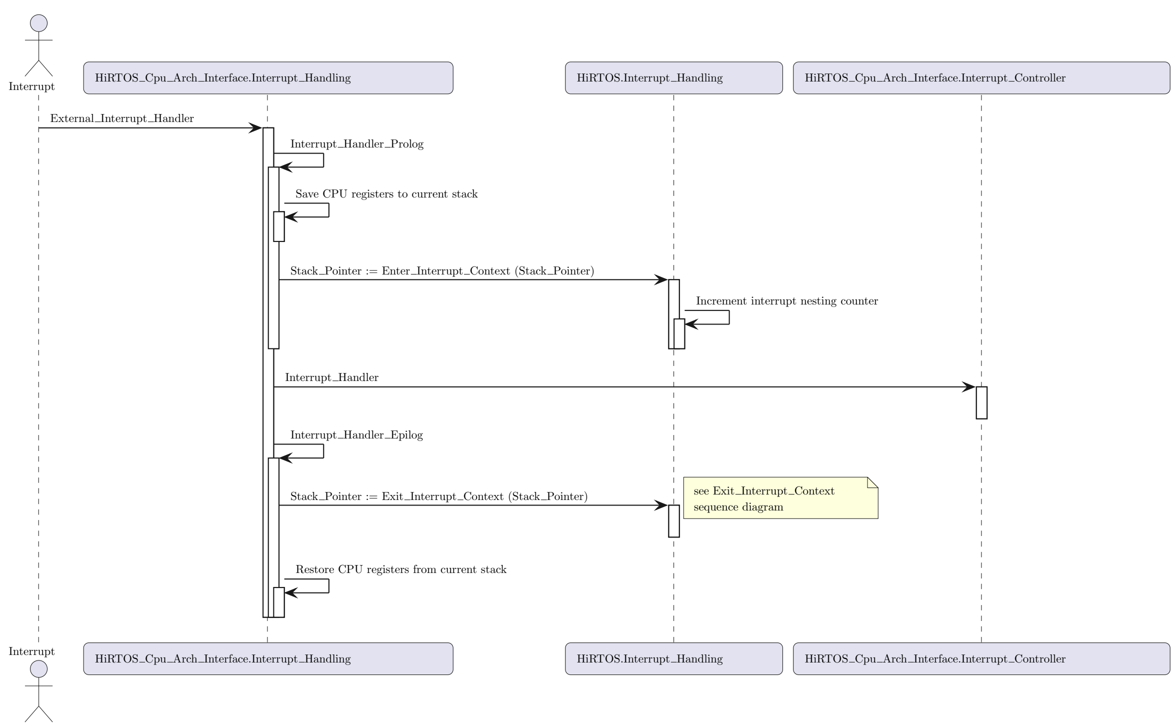

Figure 3.1: HiRTOS Asynchronous Thread Context Switch - part 1

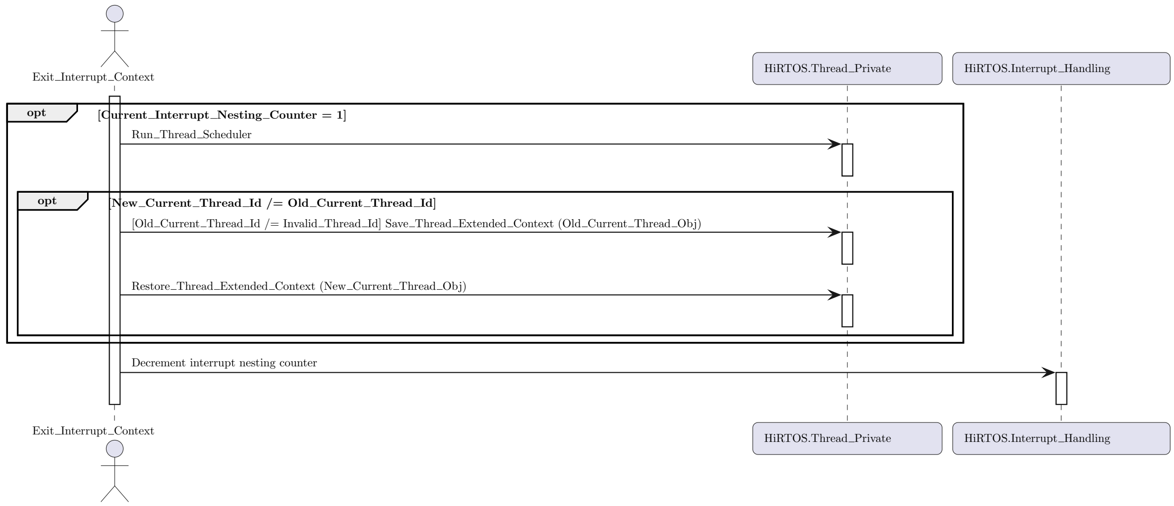

Figure 3.2: HiRTOS Asynchronous Thread Context Switch - part 2

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`

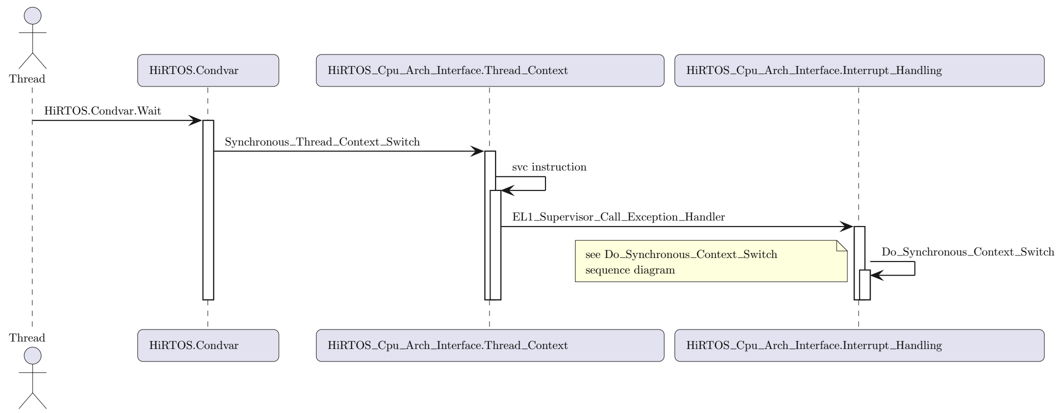

Figure 3.3: HiRTOS Synchronous Thread Context Switch - part 1

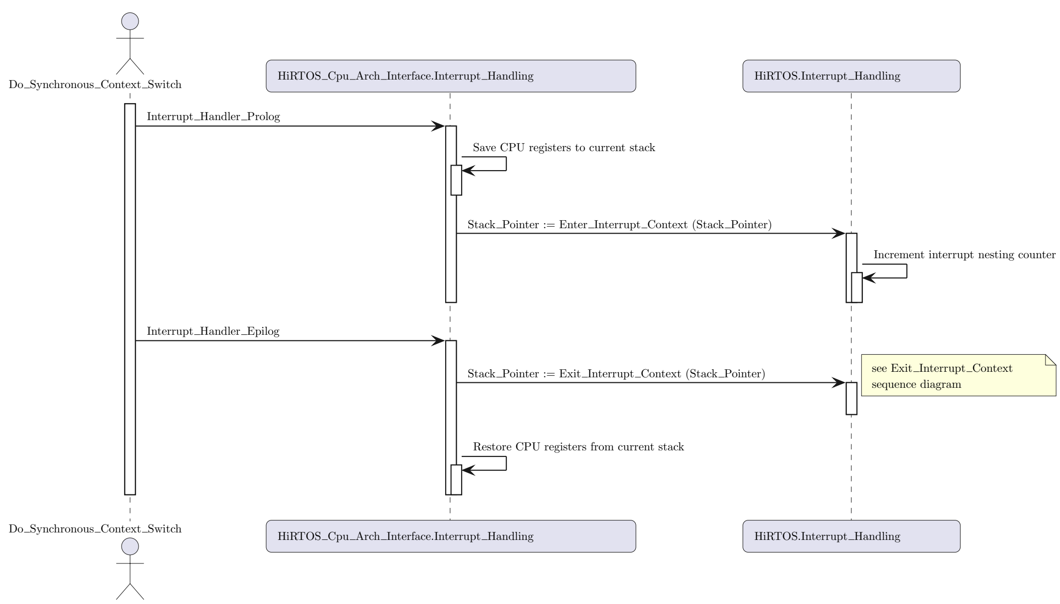

Figure 3.4: HiRTOS Synchronous Thread Context Switch - part 2

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`.

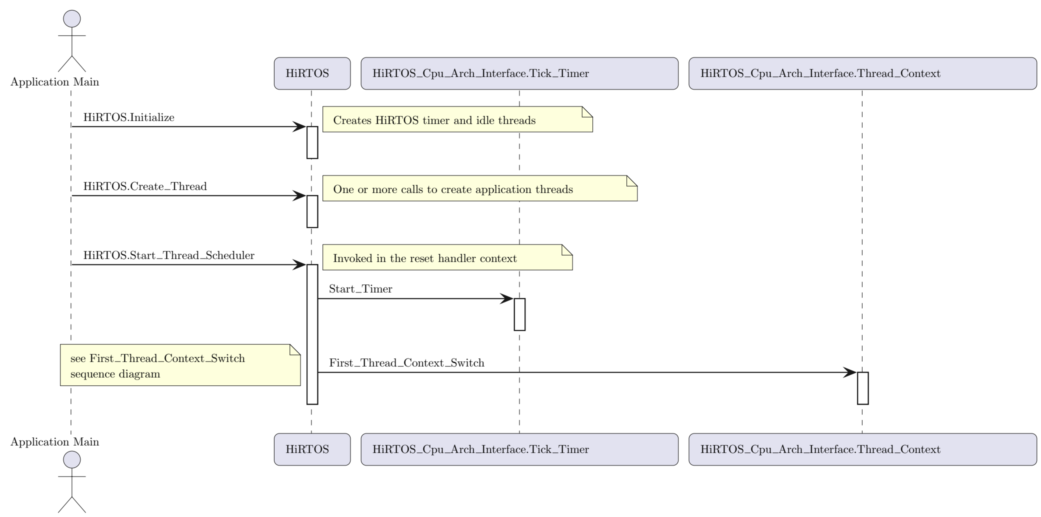

Figure 3.5: HiRTOS Initial Thread Context Switch - part 1

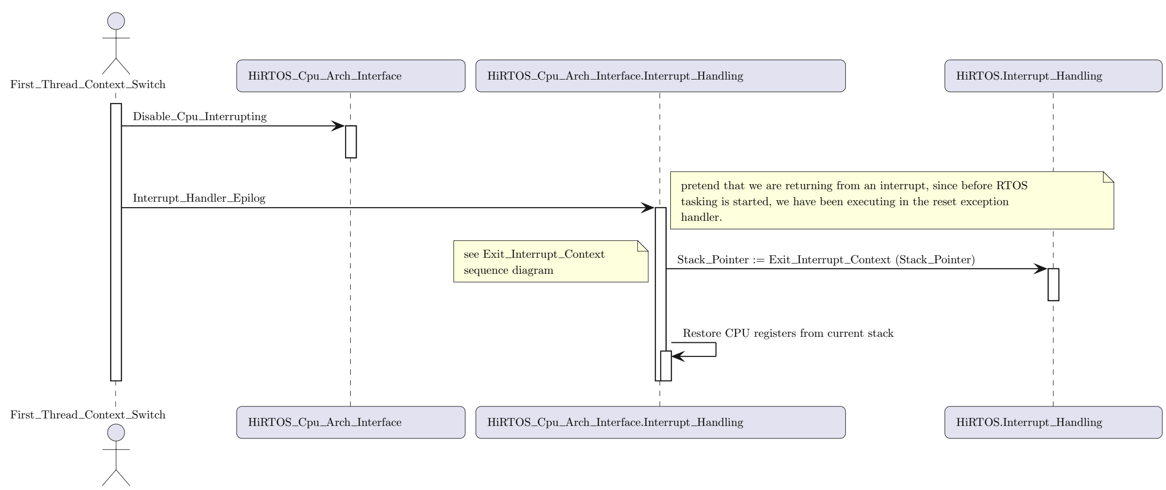

Figure 3.6: HiRTOS Initial Thread Context Switch - part 2

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:

$$
\begin{array}{l}
[PartitionIdType] \\
\#PartitionIdType = maxNumPartitionsPerCpu + 1
\end{array}
$$

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

$$
\begin{array}{l}
ValidPartitionIdType == \\
\quad PartitionIdType \setminus \{invalidPartitionId\} \\
PartitionQueueType == \operatorname{iseq} ValidPartitionIdType \\
PartitionStateType ::= partitionNotCreated \mid \\
\qquad\qquad partitionRunnable \mid partitionRunning \mid \\
\qquad\qquad partitionSuspended \\
PartitionSchedulerStateType ::= \\
\quad partitionSchedulerStopped \mid partitionSchedulerRunning \\
AddressRangeType == MemoryAddressType \times MemoryAddressType
\end{array}
$$

4.1.3 Separation Kernel State Variables

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

HiRtosSeparationKernel

$$
\begin{array}{l}
separationKernelCpuInstances : \\
\quad ValidCpuIdType \rightarrowtail\!\!\!\!\!\!\twoheadrightarrow SeparationKernelCpuInstanceType \\
zAllCreatedPartitionInstances : \mathbb{F}\, PartitionType \\
\hline
\# separationKernelCpuInstances \geq 1 \\
\forall\, i : \mathrm{dom}\, separationKernelCpuInstances \bullet \\
\quad (separationKernelCpuInstances(i)).cpuId = i \\
zAllCreatedPartitionInstances = \\
\quad \bigcup \{\, i : ValidCpuIdType \bullet \\
\qquad \mathrm{ran}\,(separationKernelCpuInstances(i)).partitions \,\} \\
\bigcap \{\, i : ValidCpuIdType \bullet \\
\quad \mathrm{ran}\,(separationKernelCpuInstances(i)).partitions \,\} = \emptyset \\
\bigcap \{\, r : \bigcup \{\, p : zAllCreatedPartitionInstances \bullet p.savedHypervisorLevelMpuRegions \,\} \bullet \\
\quad zAddressRange(r) \,\} = \emptyset
\end{array}
$$

Per-CPU Separation Kernel Instance

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

$$
\begin{array}{|l}
\hline
\textit{SeparationKernelCpuInstanceType} \\
cpuId : ValidCpuIdType \\
interruptStack : AddressRangeType \\
partitionSchedulerState : PartitionSchedulerStateType \\
currentPartitionId : PartitionIdType \\
timerTicksSinceBoot : \mathbb{N} \\
runnablePartitionsQueue : PartitionQueueType \\
partitions : ValidPartitionIdType \rightarrowtail\!\!\!\!\!\!\nrightarrow PartitionType \\
zCurrentCpuContext : CpuRegistersType \\
zCurrentHypervisorLevelMpuRegions : \mathbb{F}_1\ AddressRangeType \\
zCurrentSupervisorLevelMpuRegions : \mathbb{F}\ AddressRangeType \\
zCurrentSupervisorLevelInterruptVectorTableAddr : MemoryAddressType \\
zCurrentSupervisorLevelInterruptsEnabled : \mathbb{F}\ ValidInterruptIdType \\
zCurrentHighestInterruptPriorityDisabled : InterruptPriorityType \\
\hline
zAddressRange(interruptStack) \neq \emptyset \\
\\
currentPartitionId \neq invalidPartitionId \Rightarrow \\
\quad currentPartitionId \in \mathrm{dom}\, partitions \land \\
\quad zCurrentHypervisorLevelMpuRegions = \\
\qquad (partitions(currentPartitionId)).savedHypervisorLevelMpuRegions \land \\
\quad zCurrentSupervisorLevelInterruptVectorTableAddr = \\
\qquad (partitions(currentPartitionId)).savedInterruptVectorTableAddr \\
\\
\forall\, p : \mathrm{ran}\, partitions \bullet \\
\quad \bigcap \{\, i : p.savedHypervisorLevelMpuRegions \bullet zAddressRange(i) \,\} = \emptyset \\
\\
\forall\, i : zCurrentSupervisorLevelMpuRegions \bullet \\
\quad \exists_1\, j : zCurrentHypervisorLevelMpuRegions \bullet \\
\qquad zAddressRange(i) \subseteq zAddressRange(j) \\
\hline
\end{array}
$$

$$
\begin{array}{l}
\textbf{PartitionType} \\
\hline
\mathit{id} : \mathit{PartitionIdType} \\
\mathit{failoverPartitionId} : \mathit{PartitionIdType} \\
\mathit{state} : \mathit{PartitionStateType} \\
\mathit{timeSliceLeftUs} : \mathbb{N} \\
\mathit{savedCpuContext} : \mathit{CpuRegistersType} \\
\mathit{savedHypervisorLevelMpuRegions} : \mathbb{F}_1\ \mathit{AddressRangeType} \\
\mathit{savedSupervisorLevelMpuRegions} : \mathbb{F}\ \mathit{AddressRangeType} \\
\mathit{savedInterruptVectorTableAddr} : \mathit{MemoryAddressType} \\
\mathit{savedInterruptsEnabled} : \mathbb{F}\ \mathit{InterruptIdType} \\
\mathit{savedHighestInterruptPriorityDisabled} : \mathit{InterruptPriorityType} \\
\hline
\mathit{state} \neq \mathit{partitionNotCreated} \Rightarrow \\
\quad \mathit{id} \neq \mathit{invalidPartitionId} \\
\mathit{failoverPartitionId} \neq \mathit{invalidPartitionId} \Rightarrow \\
\quad \mathit{failoverPartitionId} \neq \mathit{id} \\
\bigcap \{\, i : \mathit{savedHypervisorLevelMpuRegions} \bullet \mathit{zAddressRange}(i) \,\} = \emptyset \\
\emptyset \notin \{\, i : \mathit{savedHypervisorLevelMpuRegions} \bullet \mathit{zAddressRange}(i) \,\} \\
\forall\, i : \mathit{savedSupervisorLevelMpuRegions} \bullet \\
\quad \exists_1\, j : \mathit{savedHypervisorLevelMpuRegions} \bullet \\
\qquad \mathit{zAddressRange}(i) \subseteq \mathit{zAddressRange}(j) \\
\hline
\end{array}
$$

4.1.4 Separation Kernel Boot-time Initialization

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

$$
\begin{array}{l}
\textbf{HiRtosSeparationKernelInitialize} \\
\hline
\mathit{HiRtosSeparationKernel}' \\
\mathit{SeparationKernelCpuInstanceInitialize} \\
\mathit{cpuId}? : \mathit{CpuIdType} \\
\hline
\mathit{cpuId}' = \mathit{cpuId}? \\
\theta \mathit{SeparationKernelCpuInstanceType}' = \mathit{separationKernelCpuInstances}'(\mathit{cpuId}?) \\
\hline
\end{array}
$$

$$
\begin{array}{l}
\textbf{SeparationKernelCpuInstanceInitialize} \\
\hline
\mathit{SeparationKernelCpuInstanceElaboration} \\
\hline
\end{array}
$$

$$
\begin{array}{|l}
\hline
\textit{SeparationKernelCpuInstanceElaboration} \\
\textit{SeparationKernelCpuInstanceType}' \\
\hline
\textit{zAddressRange}(\textit{interruptStack}') \neq \emptyset \\
\textit{partitionSchedulerState}' = \textit{partitionSchedulerStopped} \\
\textit{currentPartitionId}' = \textit{invalidPartitionId} \\
\textit{timerTicksSinceBoot}' = 0 \\
\textit{runnablePartitionsQueue}' = \langle\rangle \\
\textit{partitions}' = \emptyset \\
\hline
\end{array}
$$

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:

$$
\begin{array}{|l}
\hline
\textit{CreatePartition} \\
\Delta \textit{SeparationKernelCpuInstanceType} \\
\textit{InitializeNewPartition} \\
\hline
\textit{partitionId}! \notin \operatorname{dom} \textit{partitions} \\
\textit{partitionId}! \in \operatorname{dom} \textit{partitions}' \\
\textit{partitions}'(\textit{partitionId}!) = \textit{partitions}'(\textit{partitionId}!) \\
\textit{runnablePartitionsQueue}' = \textit{runnablePartitionsQueue} \frown \langle \textit{partitionId}! \rangle \\
\hline
\end{array}
$$

$$
\begin{array}{l}
\textit{InitializeNewPartition} \\
\hline
PartitionType' \\
partitionId! : ValidPartitionIdType \\
\hline
id' = partitionId! \\
failoverPartitionId' = invalidPartitionId \\
state' = partitionRunnable \\
savedHypervisorLevelMpuRegions' \neq \emptyset \\
savedSupervisorLevelMpuRegions' = \emptyset \\
savedInterruptsEnabled' = \emptyset \\
\hline
\end{array}
$$

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"
`http://spivey.oriel.ox.ac.uk/mike/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] FreeRTOS
`https://www.freertos.org/`

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