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

Germán Rivera
jgrivera67@gmail.com

June 14, 2023

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 samll 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 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 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 [11] and RTOS interfaces such as the CMSIS RTOS2 API [12]. 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 C/C++ interface, HiRTOS should 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*

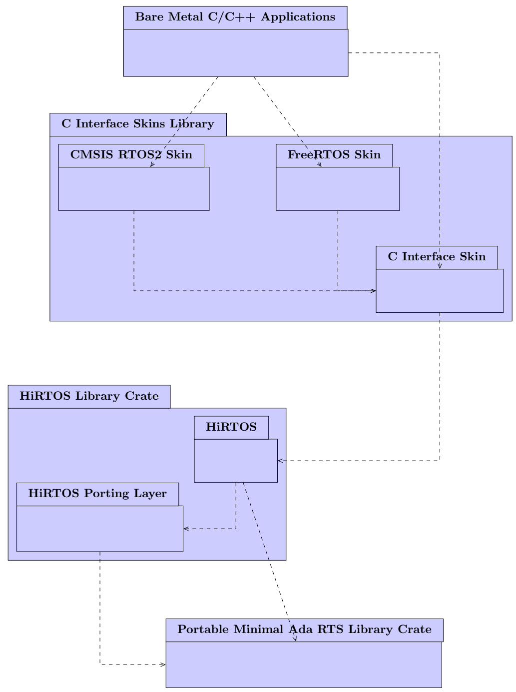

Figure 2.1: HiRTOS Code Architecture for C/C++ Applications

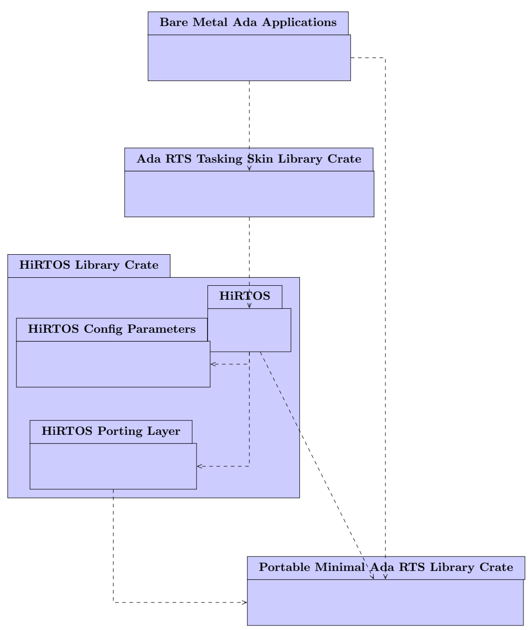

Figure 2.2: HiRTOS Code Architecture for Ada Applications

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.

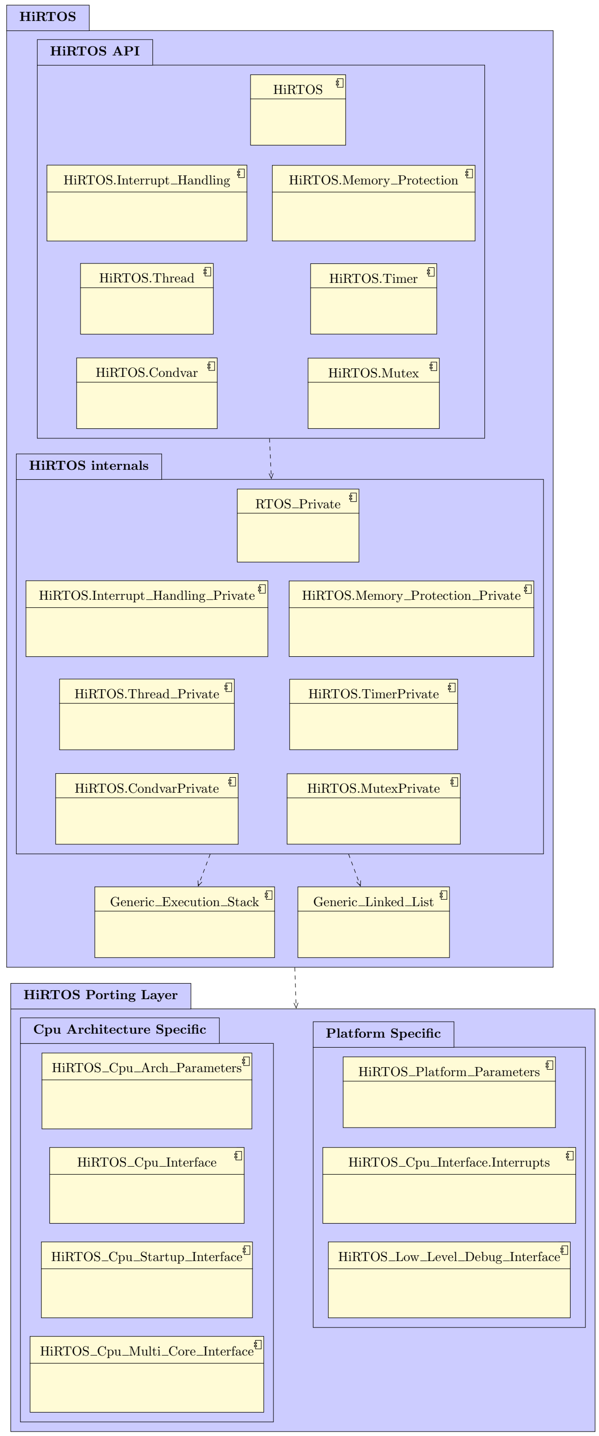

Figure 2.3: HiRTOS Code Components

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*.

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

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:

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

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

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

$$
\begin{aligned}
&\mathit{TimerTicksType} == \mathbb{N}\\
&\mathit{ThreadQueueType} == \mathrm{iseq}\ \mathit{ValidThreadIdType}\\
&\mathit{MutexListType} == \mathrm{iseq}\ \mathit{ValidMutexIdType}\\
&\mathit{TimerListType} == \mathrm{iseq}\ \mathit{ValidTimerIdType}\\
&\mathit{TimerWheelSpokeIndexType} ==\\
&\qquad 0 \mathrel{..} \mathit{numTimerWheelSpokes}\\
&\mathit{invalidTimerWheelSpokeIndex} ==\\
&\qquad \mathit{max}\ \mathit{TimerWheelSpokeIndexType}\\
&\mathit{ValidTimerWheelSpokeIndexType} ==\\
&\qquad \mathit{TimerWheelSpokeIndexType} \setminus\\
&\qquad \{\, \mathit{invalidTimerWheelSpokeIndex} \,\}
\end{aligned}
$$

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

3.1.5 *HiRTOS* Axiomatic Definitions

$$
\begin{aligned}
&\mathit{zAddressRangeSet} :\\
&\qquad (\mathit{MemoryAddressType} \times \mathit{MemoryAddressType}) \rightarrowtail\\
&\qquad \mathbb{F}_1\ \mathit{MemoryAddressType}\\
\hline
&\forall\, x, y : \mathit{MemoryAddressType} \mid\\
&\qquad (x, y) \in \mathrm{dom}\ \mathit{zAddressRangeSet} \bullet\\
&\qquad x < y \land \mathit{zAddressRangeSet}(x, y) = x \mathrel{..} y
\end{aligned}
$$

$$
\begin{aligned}
&\mathit{zCpuToISRstackAddressRange} :\\
&\qquad \mathit{ValidCpuIdType} \rightarrowtail\\
&\qquad (\mathit{MemoryAddressType} \times \mathit{MemoryAddressType})\\
\hline
&\bigcap \{\, i : \mathit{ValidCpuIdType} \bullet\\
&\mathit{zAddressRangeSet}(\mathit{zCpuToISRstackAddressRange}(i)) \,\} = \emptyset\\
&\forall\, i : \mathrm{dom}\ \mathit{zCpuToISRstackAddressRange} \bullet\\
&\#(\mathit{zAddressRangeSet}(\mathit{zCpuToISRstackAddressRange}(i))) \geq 2
\end{aligned}
$$

$$
\begin{aligned}
&\mathit{interruptPriorities} :\\
&\qquad \mathit{InterruptIdType} \rightarrow \mathit{InterruptPriorityType}
\end{aligned}
$$

3.1.6 *HiRTOS* State Variables

The *HiRtosType* singleton object type represents the internal data structures of HiRTOS. All HiRTOS objects such as threads, mutexes, condition variables and software timers are statically allocated internally by HiRTOS.

HiRtosType

$$
\begin{array}{l}
createdThreadInstances : \\
\quad ValidThreadIdType \pinj ThreadType \\
createdMutexInstances : \\
\quad ValidMutexIdType \pinj MutexType \\
createdCondvarInstances : \\
\quad ValidCondvarIdType \pinj CondvarType \\
createdTimerInstances : \\
\quad ValidTimerIdType \pinj TimerType \\
rtosCpuInstances : \\
\quad ValidCpuIdType \pinj HiRtosCpuInstanceType
\end{array}
$$

$$
\begin{array}{l}
\# rtosCpuInstances \geq 1 \\
\\
\forall i : \mathrm{dom}\, rtosCpuInstances \bullet \\
\quad (rtosCpuInstances(i)).cpuId = i \\
\\
\bigcup \{ i : ValidCpuIdType \bullet \\
\quad (rtosCpuInstances(i)).threads \} = \\
\quad createdThreadInstances \\
\\
\bigcap \{ i : ValidCpuIdType \bullet \\
\quad (rtosCpuInstances(i)).threads \} = \emptyset \\
\\
\bigcup \{ i : ValidCpuIdType \bullet \\
\quad (rtosCpuInstances(i)).mutexes \} = \\
\quad createdMutexInstances \\
\\
\bigcap \{ i : ValidCpuIdType \bullet \\
\quad (rtosCpuInstances(i)).mutexes \} = \emptyset \\
\\
\bigcup \{ i : ValidCpuIdType \bullet \\
\quad (rtosCpuInstances(i)).condvars \} = \\
\quad createdCondvarInstances \\
\\
\bigcap \{ i : ValidCpuIdType \bullet \\
\quad (rtosCpuInstances(i)).condvars \} = \emptyset \\
\\
\bigcup \{ i : ValidCpuIdType \bullet \\
\quad (rtosCpuInstances(i)).timers \} = \\
\quad createdTimerInstances \\
\\
\bigcap \{ i : ValidCpuIdType \bullet \\
\quad (rtosCpuInstances(i)).timers \} = \emptyset
\end{array}
$$

Per-CPU HiRTOS Instance

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

HiRtosCpuInstanceType

$$
\begin{array}{l}
cpuId : CpuIdType \\
threadSchedulerState : ThreadSchedulerStateType \\
currentAtomicLevel : AtomicLevelType \\
currentCpuExecutionMode : CpuExecutionModeType \\
currentThreadId : ThreadIdType \\
timerTicksSinceBoot : TimerTicksType \\
idleThreadId : ValidThreadIdType \\
tickTimerThreadId : ValidThreadIdType \\
interruptNestingLevelStack : InterruptNestingLevelStackType \\
threads : ValidThreadIdType \pinj ThreadType \\
mutexes : ValidMutexIdType \pinj MutexType \\
condvars : ValidCondvarIdType \pinj CondvarType \\
timers : ValidTimerIdType \pinj TimerType \\
runnableThreadsQueue : ThreadPriorityQueueType \\
timerWheel : TimerWheelType \\
zCpuInterruptMaskingState : CpuInterruptMaskingStateType \\
zCpuPrivilege : CpuPrivilegeType \\
zMemoryProtectionState : MemoryProtectionStateType
\end{array}
$$

$$
\begin{array}{l}
\{ idleThreadId, tickTimerThreadId \} \subseteq \mathrm{dom}\, threads \\[1ex]
tickTimerThreadId \neq idleThreadId \\[1ex]
threadSchedulerState = threadSchedulerRunning \Rightarrow \\
(\forall t : \mathrm{ran}(\{ currentThreadId \} \ndres threads) \bullet \\
\quad t.currentPriority < (threads(currentThreadId)).currentPriority) \\[1ex]
zCpuInterruptMaskingState = cpuInterruptsEnabled \Leftrightarrow \\
\quad currentAtomicLevel > atomicLevelNoInterrupts \\[1ex]
zCpuInterruptMaskingState = cpuInterruptsDisabled \Rightarrow zCpuPrivilege = cpuPrivileged \\[1ex]
currentAtomicLevel < atomicLevelNone \Rightarrow zCpuPrivilege = cpuPrivileged \\[1ex]
\bigcap \{ t : \mathrm{ran}\, threads \bullet \{ t.builtinCondvarId \} \} = \emptyset \\[1ex]
\bigcap \{ t : \mathrm{ran}\, threads \bullet \{ t.builtinTimerId \} \} = \emptyset \\[1ex]
\forall t : \mathrm{ran}\, threads \bullet (t.id = (threads^{\sim})(t) \land \\
(t.ownedMutexes \neq \emptyset \Rightarrow \\
t.currentPriority = max \{ m : \mathrm{ran}\, t.ownedMutexes \bullet (mutexes(m)).ceilingPriority \})) \\[1ex]
\forall m : \mathrm{ran}\, mutexes \bullet m.id = mutexes^{\sim}(m) \\[1ex]
\forall c : \mathrm{ran}\, condvars \bullet c.id = condvars^{\sim}(c) \\[1ex]
\forall ti : \mathrm{ran}\, timers \bullet ti.id = timers^{\sim}(ti) \\[1ex]
\forall p : ValidThreadPriorityType \bullet \\
\quad \forall threadId : \mathrm{ran}(runnableThreadsQueue.threadQueues(p)) \bullet \\
\qquad (threads(threadId)).currentPriority = p
\end{array}
$$

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

$$
\begin{array}{|l}
\textit{InterruptNestingLevelStackType} \\
\hline
\textit{interruptNestingLevels} : \\
\quad \textit{ActiveInterruptNestingCounterType} \rightarrowtail \\
\quad \textit{InterruptNestingLevelType} \\
\textit{currentInterruptNestingCounter} : \\
\quad \textit{InterruptNestingCounterType} \\
\textit{zCpuId} : \textit{ValidCpuIdType} \\
\hline
\forall x : \textit{ActiveInterruptNestingCounterType} \bullet \\
\quad (\textit{interruptNestingLevels}(x)).\textit{interruptNestingCounter} = x \\
\quad \wedge \\
\quad (\textit{interruptNestingLevels}(x)).\textit{savedStackPointer} \in \\
\quad \textit{zAddressRangeSet}(\textit{zCpuToISRstackAddressRange}(\textit{zCpuId})) \\
\\
\operatorname{dom} \textit{interruptNestingLevels} = \\
1 \mathrel{..} \textit{currentInterruptNestingCounter} \\
\hline
\end{array}
$$

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

$$
\begin{array}{|l}
\textit{TimerWheelType} \\
\hline
\textit{wheelSpokesHashTable} : \\
\quad \textit{ValidTimerWheelSpokeIndexType} \rightarrow \mathbb{F}\ \textit{TimerIdType} \\
\textit{currentWheelSpokeIndex} : \textit{ValidTimerWheelSpokeIndexType} \\
\hline
\bigcap \{ i : \textit{ValidTimerWheelSpokeIndexType} \bullet \\
\quad \textit{wheelSpokesHashTable}(i) \} = \emptyset \\
\hline
\end{array}
$$

$$
\begin{array}{|l}
\hline
\textit{TimerType} \\
id : TimerIdType \\
\hline
\end{array}
$$

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

$$
\begin{array}{|l}
\hline
\textit{CondvarType} \\
id : \textit{CondvarIdType} \\
\hline
\end{array}
$$

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

3.2 HiRTOS Boot-time Initialization

3.2.1 HiRTOS Elaboration-time Initialization

On boot, before the `HiRTOS.Initialize` *HiRTOS* API is called on any CPU core, the global state of *HiRTOS* is as follows:

$$
\begin{array}{|l}
\hline
\textit{HiRtosInitialState} \\
\textit{HiRtosType}' \\
\hline
\textit{createdThreadInstances}' = \emptyset \\
\textit{createdMutexInstances}' = \emptyset \\
\textit{createdCondvarInstances}' = \emptyset \\
\textit{createdTimerInstances}' = \emptyset \\
\hline
\end{array}
$$

HiRtosCpuInstanceInitialState

$HiRtosCpuInstanceType'$

$threadSchedulerState' = threadSchedulerStopped$

$currentThreadId' = invalidThreadId$

$currentAtomicLevel' = atomicLevelNone$

$currentCpuExecutionMode' = cpuExecutingResetHandler$

$idleThreadId' = invalidThreadId$

$tickTimerThreadId' = invalidThreadId$

$threads' = \emptyset$

$condvars' = \emptyset$

$timers' = \emptyset$

$timerTicksSinceBoot' = 0$

$\forall p : ValidThreadPriorityType \bullet$
$runnableThreadsQueue'.threadQueues(p) = \emptyset$

InterruptNestingLevelStackInitialState

$InterruptNestingLevelStackType'$

$currentInterruptNestingCounter' = 1$

InterruptNestingLevelInitialState

$InterruptNestingLevelType'$

$interruptId' = invalidInterruptId$

$interruptNestingCounter' = 0$

$savedStackPointer' = nullAddress$

$atomicLevel' = atomicLevelNone$

TimerWheelInitialState

$TimerWheelType'$

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

$currentWheelSpokeIndex' =$
$\quad min\ ValidTimerWheelSpokeIndexType$

$$
\begin{array}{|l}
\hline
\textit{ThreadTypeInitialState} \\
\textit{ThreadType}' \\
\hline
id' = invalidThreadId \\
builtinTimerId' = invalidTimerId \\
builtinCondvarId' = invalidCondvarId \\
state' = threadNotCreated \\
atomicLevel' = atomicLevelNone \\
\hline
\end{array}
$$

$$
\begin{array}{|l}
\hline
\textit{MutexTypeInitialState} \\
\textit{MutexType}' \\
\hline
id' = invalidMutexId \\
ownerThreadId' = invalidThreadId \\
recursiveCount' = 0 \\
ceilingPriority' = invalidThreadPriority \\
\forall p : ValidThreadPriorityType \bullet \\
\qquad waitingThreadsQueue'.threadQueues(p) = \emptyset \\
\hline
\end{array}
$$

3.2.2 HiRTOS Runtime-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:

$$
\begin{array}{|l}
\hline
\textit{HiRtosInitialize} \\
\Delta \textit{HiRtosType} \\
\Delta \textit{HiRtosCpuInstanceType} \\
\textit{cpuId}? : \textit{CpuIdType} \\
\hline
\theta \textit{HiRtosCpuInstanceType} = \textit{rtosCpuInstances}(\textit{cpuId}?) \\
\theta \textit{HiRtosCpuInstanceType}' = \textit{rtosCpuInstances}'(\textit{cpuId}?) \\
(\textit{rtosCpuInstances}'(\textit{cpuId}?)).\textit{cpuId} = \textit{cpuId}? \\
\textit{idleThreadId}' \neq \textit{invalidThreadId} \\
\textit{tickTimerThreadId}' \neq \textit{invalidThreadId} \\
\textit{tickTimerThreadId}' \neq \textit{idleThreadId}' \\
\operatorname{dom} \textit{threads}' = \{\, \textit{idleThreadId}', \textit{tickTimerThreadId}' \,\} \\
\operatorname{dom} \textit{condvars}' = \\
\{\, (\textit{threads}'(\textit{idleThreadId}')).\textit{builtinCondvarId}, \\
(\textit{threads}'(\textit{tickTimerThreadId}')).\textit{builtinCondvarId} \,\} \\
\operatorname{dom} \textit{timers}' = \\
\{\, (\textit{threads}'(\textit{idleThreadId}')).\textit{builtinTimerId}, \\
(\textit{threads}'(\textit{tickTimerThreadId}')).\textit{builtinTimerId} \,\} \\
\textit{interruptNestingLevelStack}'.\textit{zCpuId} = \textit{cpuId}' \\
\textit{zCpuInterruptMaskingState}' = \textit{cpuInterruptsEnabled} \\
\textit{zCpuPrivilege}' = \textit{cpuUnprivileged} \\
\textit{zMemoryProtectionState}' = \textit{memoryProtectionOn} \\
\textit{runnableThreadsQueue}'.\textit{threadQueues}(\textit{min ValidThreadPriorityType}) \\
\quad = \langle \textit{idleThreadId}' \rangle \\
\textit{runnableThreadsQueue}'.\textit{threadQueues}(\textit{max ValidThreadPriorityType}) \\
\quad = \langle \textit{tickTimerThreadId}' \rangle \\
\hline
\end{array}
$$

3.3 HiRTOS Callable Services

3.3.1 *HiRTOS* Mutex Operations

Create a new mutex

A mutex can be created by calling `HiRTOS.Mutex.Create`. Mutexes are allocated from a pool of mutex objects shared among all CPU cores:

$$
\begin{array}{|l}
\multicolumn{1}{l}{\textit{CreateMutex}} \\
\Delta \mathit{HiRtosType} \\
\Delta \mathit{HiRtosCpuInstanceType} \\
\mathit{InitializeNewMutex} \\
\mathit{cpuId?} : \mathit{CpuIdType} \\
\hline
\theta \mathit{HiRtosCpuInstanceType} = \mathit{rtosCpuInstances}(\mathit{cpuId?}) \\
\theta \mathit{HiRtosCpuInstanceType}' = \mathit{rtosCpuInstances}'(\mathit{cpuId?}) \\
\theta \mathit{MutexType}' = \mathit{createdMutexInstances}'(\mathit{mutexId!}) \\
\mathit{mutexId!} \notin \operatorname{dom} \mathit{createdMutexInstances} \\
\mathit{mutexId!} \notin \operatorname{dom} \mathit{mutexes} \\
\mathit{mutexId!} \in \operatorname{dom} \mathit{createdMutexInstances}' \\
\mathit{mutexes}'(\mathit{mutexId!}) = \mathit{createdMutexInstances}'(\mathit{mutexId!}) \\
\hline
\end{array}
$$

$$
\begin{array}{|l}
\multicolumn{1}{l}{\textit{InitializeNewMutex}} \\
\Delta \mathit{MutexType} \\
\mathit{ceilingPriority?} : \mathit{ThreadPriorityType} \\
\mathit{mutexId!} : \mathit{ValidMutexIdType} \\
\hline
\mathit{mutexId!} \neq \mathit{invalidMutexId} \\
\mathit{id}' = \mathit{mutexId!} \\
\mathit{ceilingPriority}' = \mathit{ceilingPriority?} \\
\hline
\end{array}
$$

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`:

$$
\begin{array}{|l}
\multicolumn{1}{l}{\textit{CreatedMutexMutableOperation}} \\
\hline
\Delta \mathit{HiRtosType} \\
\Delta \mathit{HiRtosCpuInstanceType} \\
\Delta \mathit{MutexType} \\
\mathit{cpuId}? : \mathit{CpuIdType} \\
\mathit{mutexId}? : \mathit{ValidMutexIdType} \\
\hline
\theta \mathit{HiRtosCpuInstanceType} = \mathit{rtosCpuInstances}(\mathit{cpuId}?) \\
\theta \mathit{HiRtosCpuInstanceType}' = \mathit{rtosCpuInstances}'(\mathit{cpuId}?) \\
\theta \mathit{MutexType} = \mathit{mutexes}(\mathit{mutexId}?) \\
\theta \mathit{MutexType}' = \mathit{mutexes}'(\mathit{mutexId}?) \\
\hline
\end{array}
$$

Acquire a mutex

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

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

$$
\begin{array}{l}
\textit{WaitOnUnavailableMutex} \\
\hline
\textit{CreatedMutexMutableOperation} \\
\hline
\textit{ownerThreadId} \neq \textit{invalidThreadId} \\[1ex]
\textit{currentThreadId} \neq \textit{ownerThreadId} \\[1ex]
\textit{currentThreadId}' \neq \textit{currentThreadId} \\[1ex]
(\mathbf{let}\ \textit{oldCurrentPriority} == \\
\quad (\textit{threads}(\textit{currentThreadId})).\textit{currentPriority} \bullet \\
(\textit{ceilingPriority} = \textit{invalidThreadPriority} \land \\
\textit{oldCurrentPriority} > \\
\quad (\textit{threads}(\textit{ownerThreadId})).\textit{currentPriority}) \Rightarrow \\
(\textit{threads}'(\textit{ownerThreadId})).\textit{currentPriority} = \\
\quad \textit{oldCurrentPriority} \\
\land \\
\textit{currentThreadId}' \neq \textit{ownerThreadId} \Rightarrow \\
((\textit{threads}'(\textit{currentThreadId}')).\textit{currentPriority} \geq \\
\quad (\textit{threads}'(\textit{ownerThreadId})).\textit{currentPriority} \lor \\
(\textit{threads}'(\textit{ownerThreadId})).\textit{state} = \textit{threadBlocked}) \\
\land \\
(\textit{threads}'(\textit{currentThreadId})).\textit{state} = \textit{threadBlocked} \\
\land \\
\textit{currentThreadId} \in \\
\quad \operatorname{ran}(\textit{waitingThreadsQueue}'.\textit{threadQueues}(\textit{oldCurrentPriority}))) \\[1ex]
\textit{ceilingPriority} \neq \textit{invalidThreadPriority} \Rightarrow \\
((\textit{threads}(\textit{currentThreadId})).\textit{currentPriority} \leq \\
\quad (\textit{threads}'(\textit{ownerThreadId})).\textit{currentPriority} \land \\
(\textit{threads}'(\textit{ownerThreadId})).\textit{state} = \textit{threadBlocked}) \\
\hline
\end{array}
$$

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

Release a mutex

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

$$
\begin{array}{l}
\textit{ReleaseMutex} \\
\hline
\textit{CreatedMutexMutableOperation} \\
\hline
\textit{mutexId}? = \textit{head}\ (\textit{threads}(\textit{currentThreadId})).\textit{ownedMutexes} \\[1ex]
(\textit{threads}'(\textit{currentThreadId})).\textit{ownedMutexes} = \textit{tail}\ (\textit{threads}(\textit{currentThreadId})).\textit{ownedMutexes} \\
\hline
\end{array}
$$

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] FreeRTOS
`https://www.freertos.org/`

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