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

Germán Rivera
jgrivera67@gmail.com

February 24, 2023

Contents

Chapter 1

Introduction

This document describes the design of *HiRTOS* (*"High Integrity"* RTOS), a real-time operating system kernel that supports multi-core systems and that is specifically designed for high integrity applications. The design is presented using the Z notation [3, 4].

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 [5], a Z type-checker, that catches Z type mismatches in predicates.

The code of *HiRTOS* is written in SPARK Ada [7], a high integrity subset of the Ada programming language. SPARK Ada code can be formally verified at compile-time with the `gnatprove` tool [8].

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.

1.2 Major Design Decisions

- 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.

- For API simplicity, inspired by the thread synchronization primitives of the C11 standard library [1], mutexes and condition variables are the only real synchronization primitives in *HiRTOS*. Other synchronization primitives such as semaphores, event flags and message queues can be implemented on top of them.

- Unlike stadanrd mutexes, *HiRTOS* mutexes have priorities to support the priority ceiling protocol [2].

- *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.

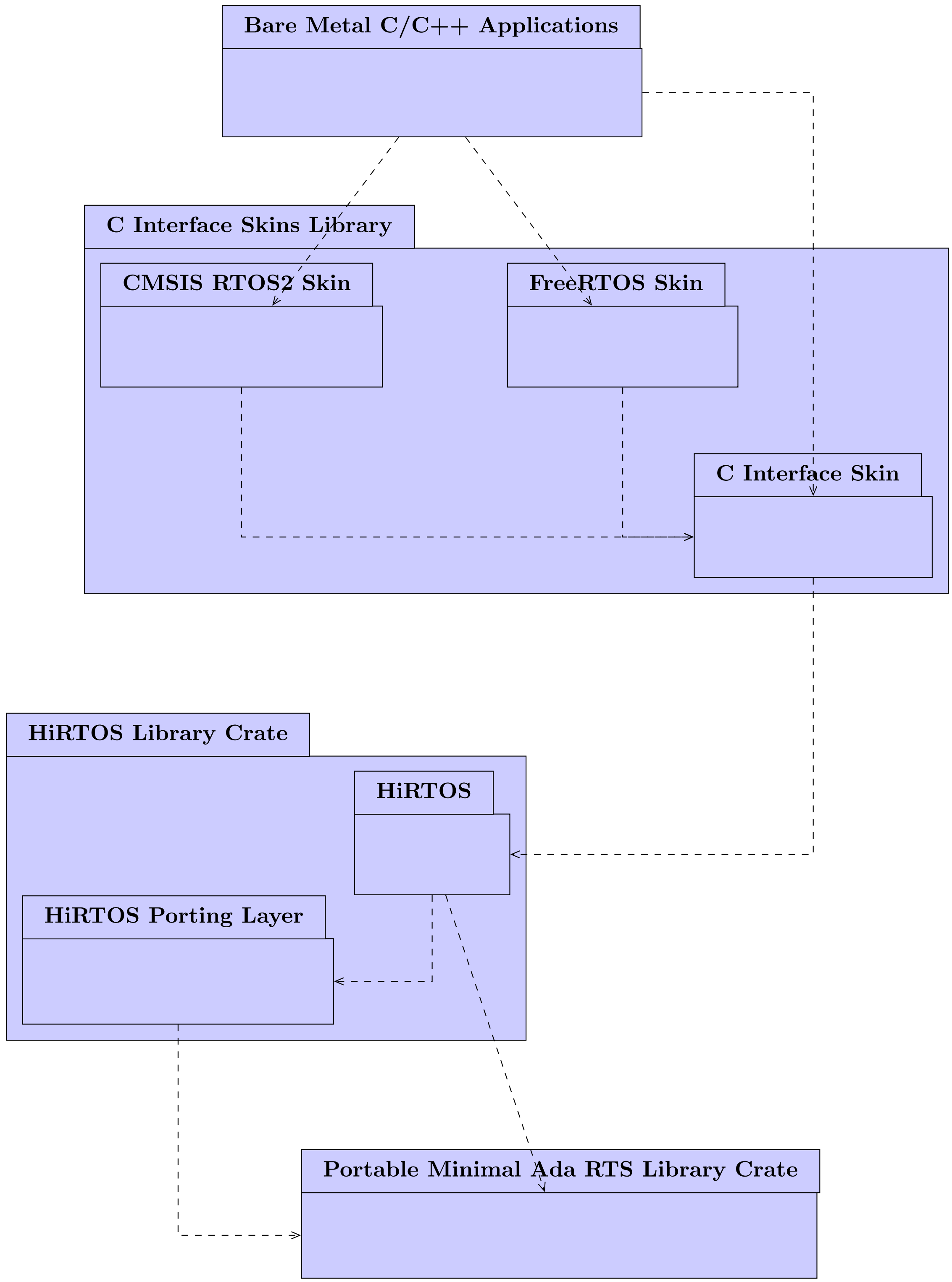
- In a multi-core platform, there is one *HiRTOS* instance per CPU Core. Each *HiRTOS* instance is independent of each other. No resources are shared between *HiRTOS* instances. No communication between CPU cores is supported by *HiRTOS*, so that the *HiRTOS* API can stay the same for both single-core 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.

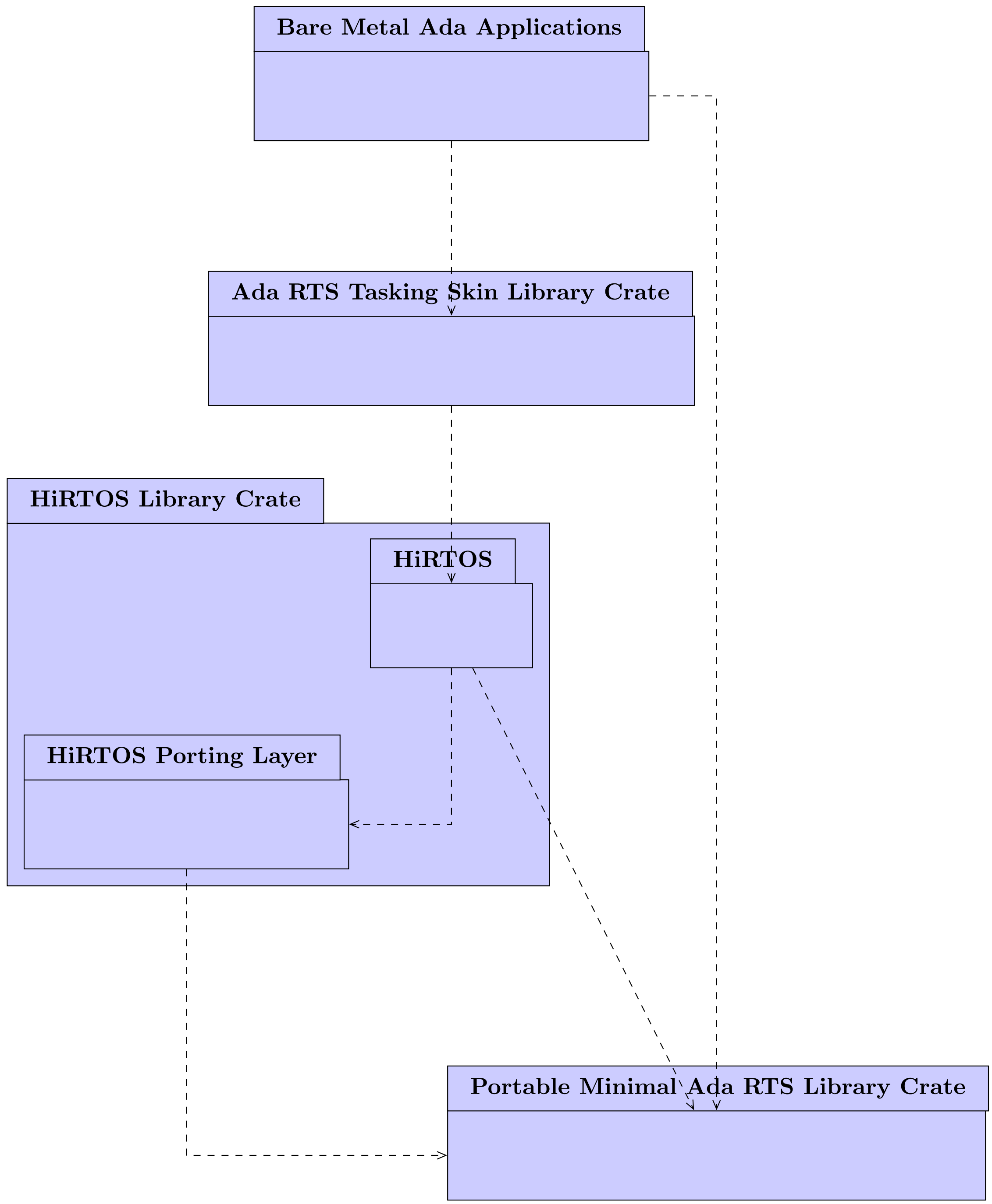
- Threads are bound to the CPU core in which they were created, for the lifetime of the thread. That is, no thread migration between CPU cores is supported.

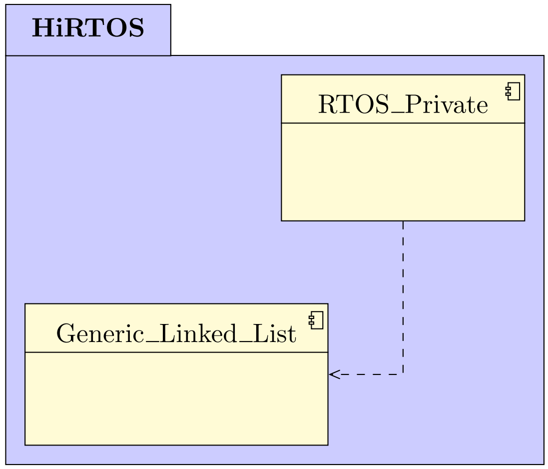
- All RTOS objects such as threads, mutexes and condition variables are allocated internally by *HiRTOS* from statically allocated internal object pools. These object pools are just RTOS-private global arrays of the corresponding RTOS object types, sized at compile time via configuration parameters, 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.

1.3 Highl-level Architecture




Bare Metal Ada Applications
Ada RTS Tasking Skin Library Crate
HiRTOS Library Crate
HiRTOS
HiRTOS Porting Layer
Portable Minimal Ada RTS Library Crate


HiRTOS
RTOS_Private
Generic_Linked_List

Chapter 2

HiRTOS Z Specification

2.1 *HiRTOS* Configuration Parameters

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

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

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

2.2 *HiRTOS* Target Platform Parameters

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

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

2.3 *HiRTOS* Primitive Types

$CpuIdType == 0 \mathrel{.\,.} maxNumCpus$
$invalidCpuId == maxNumCpus$
$ValidCpuIdType == CpuIdType \setminus \{invalidCpudId\}$
$MemoryAddressType == minMemoryAddress \mathrel{.\,.} maxMemoryAddress$
$nullAddress == 0$
$ThreadIdType == 0 \mathrel{.\,.} maxNumThreads$
$invalidThreadId == maxNumThreads$
$ValidThreadIdType == ThreadIdType \setminus \{invalidThreadId\}$
$ThreadPriorityType == 0 \mathrel{.\,.} numThreadPriorities$
$invalidThreadPriority == numThreadPriorities$
$ValidThreadPriorityType == ThreadPriorityType \setminus \{invalidThreadPriority\}$
$MutexIdType == 0 \mathrel{.\,.} maxNumMutexes$
$invalidMutexId == maxNumMutexes$
$ValidMutexIdType == MutexIdType \setminus \{invalidMutexId\}$
$CondvarIdType == 0 \mathrel{.\,.} maxNumCondvars$
$invalidCondvarId == maxNumCondvars$
$ValidCondvarIdType == CondvarIdType \setminus \{invalidCondvarId\}$
$TimerIdType == 0 \mathrel{.\,.} maxNumTimers$
$invalidTimerId == maxNumTimers$
$ValidTimerIdType == TimerIdType \setminus \{invalidTimerId\}$
$InterruptPrioirtyType == 0 \mathrel{.\,.} numInterruptPriorities$
$invalidInterruptPriority == numInterruptPriorities$
$ValidInterruptPriorityType == InterruptPriorityType \setminus \{invalidInterruptPriority\}$
$AtomicLevelType == 0 \mathrel{.\,.} numInterruptPriorities + 1$
$atomicLevelNoInterrupts == min\ AtomicLevelType$
$atomicLevelSingleThread == max\ AtomicLevelType - 1$
$atomicLevelNone == max\ AtomicLevelType$
$InterruptNestingCounterType == 0 \mathrel{.\,.} numInterruptPriorities$
$ActiveInterruptNestingCounterType == InterruptNestingCounterType \setminus \{ 0 \}$
$CpuInterruptMaskingStateType ::= cpuInterruptsEnabled \mid cpuInterruptsDisabled$
$CpuPrivilegeType ::= cpuPrivileged \mid cpuUnprivileged$
$MemoryProtectionStateType ::= memoryProtectionOn \mid memoryProtectionOff$
$CpuExecutionModeType ::= cpuExecutingResetHandler \mid cpuExecutingInterruptHandler \mid cpuExecutingThread$
$ThreadStateType ::= threadNotCreated \mid threadRunnable \mid threadRunning \mid threadInterrupted \mid threadBlocked$
$HiRtosStateType ::= threadSchedulerStopped \mid threadSchedulerRunning$
$TimerTicksType == \mathbb{N}$
$ThreadQueueType == \mathrm{iseq}\ ValidThreadIdType$
$MutexListType == \mathrm{iseq}\ ValidMutexIdType$
$TimerListType == \mathrm{iseq}\ ValidTimerIdType$
$TimerWheelSpokeIndexType == 0 \mathrel{.\,.} numTimerWheelSpokes$
$invalidTimerWheelSpokeIndex == max\ TimerWheelSpokeIndexType$
$ValidTimerWheelSpokeIndexType == TimerWheelSpokeIndexType \setminus \{ invalidTimerWheelSpokeIndex \}$
$PerCpuThreadSetType == \mathbb{F}_1\ ValidThreadIdType$
$PerCpuMutexSetType == \mathbb{F}\ ValidMutexIdType$
$PerCpuCondvarSetType == \mathbb{F}_1\ ValidCondvarIdType$

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

2.4 *HiRTOS* Axiomatic Definitions

$$
\begin{array}{|l}
zThreadInstances : ValidThreadIdType \rightarrowtail ThreadType \\
zMutexInstances : ValidMutexIdType \rightarrowtail MutexType \\
zCondvarInstances : ValidCondvarIdType \rightarrowtail CondvarType \\
zTimerInstances : ValidTimerIdType \rightarrowtail TimerType \\
zRtosCpuInstances : ValidCpuIdType \rightarrowtail HiRtosCpuInstanceType \\
zCpuToISRstackAddressRange : ValidCpuIdType \rightarrowtail \mathbb{F}_1\ MemoryAddressType \\
\hline
\forall i : \mathrm{dom}\ rtosCpuInstances \bullet rtosCpuInstances(i).cpuId = i \\
\bigcap \{ i : ValidCpuIdType \bullet \\
\quad zCpuToISRstackAddressRange(i) \} = \emptyset
\end{array}
$$

$$
\begin{array}{|l}
zGetHighestPriorityThread : ValidCpuIdType \rightarrowtail ThreadIdType \\
\hline
\forall cpuId : ValidCpuIdType \bullet \\
\quad (\mathbf{let}\ threadId == zGetHighestPriorityThread(cpuId) \bullet \\
\quad\quad threadId \in zRtosCpuInstances(cpuId).allThreads \land \\
\quad\quad (\forall i : zRtosCpuInstances(cpuId).allThreads \setminus \{ threadId \} \bullet \\
\quad\quad\quad zThreadInstances(i).priority < zThreadInstances(threadId).priority))
\end{array}
$$

$$
\begin{array}{|l}
interruptPriorities : InterruptIdType \rightarrow InterruptPrioirtyType
\end{array}
$$

2.5 ***HiRTOS*** State Variables

$$
\begin{array}{|l}
\hline
\textit{HiRtosType} \\
\textit{createdThreadInstances} : \textit{ValidThreadIdType} \pinj \textit{ThreadType} \\
\textit{createdMutexInstances} : \textit{ValidMutexIdType} \pinj \textit{MutexType} \\
\textit{createdCondvarInstances} : \textit{ValidCondvarIdType} \pinj \textit{CondvarType} \\
\textit{createdTimerInstances} : \textit{ValidTimerIdType} \pinj \textit{TimerType} \\
\textit{rtosCpuInstances} : \textit{ValidCpuIdType} \pinj \textit{HiRtosCpuInstanceType} \\
\hline
\textit{createdThreadInstances} \subseteq \textit{zThreadInstances} \\
\textit{createdMutexInstances} \subseteq \textit{zMutexdInstances} \\
\textit{createdCondvarInstances} \subseteq \textit{zCondvarInstances} \\
\textit{createdTimerInstances} \subseteq \textit{zTimerInstances} \\
\textit{rtosCpuInstances} = \textit{zRtosCpuInstances} \\
\bigcup \{\, i : \textit{ValidCpuIdType} \bullet \\
\quad \textit{rtosCpuInstances}(i).\textit{allThreads} \,\} = \textit{createdThreadInstances} \\
\bigcap \{\, i : \textit{ValidCpuIdType} \bullet \\
\quad \textit{rtosCpuInstances}(i).\textit{allThreads} \,\} = \emptyset \\
\bigcup \{\, i : \textit{ValidCpuIdType} \bullet \\
\quad \textit{rtosCpuInstances}(i).\textit{allMutexes} \,\} = \textit{createdMutexInstances} \\
\bigcap \{\, i : \textit{ValidCpuIdType} \bullet \\
\quad \textit{rtosCpuInstances}(i).\textit{allMutexes} \,\} = \emptyset \\
\bigcup \{\, i : \textit{ValidCpuIdType} \bullet \\
\quad \textit{rtosCpuInstances}(i).\textit{allCondvars} \,\} = \textit{createdCondvarInstances} \\
\bigcap \{\, i : \textit{ValidCpuIdType} \bullet \\
\quad \textit{rtosCpuInstances}(i).\textit{allCondvars} \,\} = \emptyset \\
\bigcup \{\, i : \textit{ValidCpuIdType} \bullet \\
\quad \textit{rtosCpuInstances}(i).\textit{allTimers} \,\} = \textit{createdTimerInstances} \\
\bigcap \{\, i : \textit{ValidCpuIdType} \bullet \\
\quad \textit{rtosCpuInstances}(i).\textit{allTimers} \,\} = \emptyset \\
\hline
\end{array}
$$

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

HiRtosCpuInstanceType

$$
\begin{array}{l}
cpuId : CpuIdType \\
threadSchedulerState : ThreadSchedulerStateType \\
currentAtomicLevel : AtomicLevelType \\
currentCpuExecutionMode : CpuExecutionModeType \\
currentThreadId : ThreadIdType \\
timerTicksSinceBoot : TimerTicksType \\
idleThreadId : ValidThreadIdType \\
tickTimerThreadId : ValidThreadIdType \\
interruptNestingLevelStack : InterruptNestingLevelStackType \\
allThreads : PerCpuThreadSetType \\
allMutexes : PerCpuMutexSetType \\
allCondvars : PerCpuCondvarSetType \\
allTimers : PerCpuTimerSetType \\
runnableThreadQueues : ValidThreadPriorityIdType \rightarrowtail ThreadQueueType \\
timerWheel : TimerWheelType \\
zCpuInterruptMaskingState : CpuInterruptMaskingStateType \\
zCpuPrivilege : CpuPrivilegeType \\
zMemoryProtectionState : MemoryProtectionStateType
\end{array}
$$

$$
\begin{array}{l}
\{ idleThreadId, tickTimerThreadId \} \subseteq allThreads \\[1ex]
tickTimerThreadId \neq idleThreadId \\[1ex]
threadSchedulerState = threadSchedulerRunning \Rightarrow \\
\quad currentThreadId = zGetHighestPriorityThread(cpuId) \\[1ex]
zCpuInterruptMaskingState = cpuInterruptsEnabled \Leftrightarrow \\
\quad currentAtomicLevel > AtomicLevelNoInterrupts \\[1ex]
zCpuInterruptMaskingState = cpuInterruptsDisabled \Rightarrow \\
\quad zCpuPrivilege = cpuPrivileged \\[1ex]
currentAtomicLevel < AtomicLevelNone \Rightarrow zCpuPrivilege = cpuPrivileged \\[1ex]
\bigcap \{ i : allThreads \bullet \{ zThreadInstances(i).builtinCondvarId \} \} = \emptyset \\[1ex]
\bigcap \{ i : allThreads \bullet \{ zThreadInstances(i).delayTimerId \} \} = \emptyset \\[1ex]
\forall p : ValidThreadPrioirtyType \bullet \\
\quad \forall t : \mathrm{ran}\ runnableThreadQueues(p) \bullet t.priority = p
\end{array}
$$

InterruptNestingLevelStackType

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

InterruptNestingLevelType

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

TimerWheelType

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

TimerType

ThreadType

CondvarType

MutexType

2.6 *HiRTOS* Initialization

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

HiRtosInitialState

$$HiRtosType'$$

$$createdThreadInstances' = \emptyset$$

$$createdMutexInstances' = \emptyset$$

$$createdCondvarInstances' = \emptyset$$

$$createdTimerInstances' = \emptyset$$

When `HiRTOS.Initialize` is called for a given CPU core, the idle thread and the tick timer thread for that CPU are created. The initial state of the *HiRTOS* instance for that CPU is as follows:

$$
\begin{array}{l}
\textbf{HiRtosCpuInstanceInitialState} \\
\hline
HiRtosCpuInstanceType' \\
\hline
threadSchedulerState' = threadSchedulerStopped \\
currentThreadId' = invalidThreadId \\
currentAtomicLevel' = AtomicLevelNone \\
currentCpuExecutionMode' = cpuExecutingResetHandler \\
idleThreadId' \neq invalidThreadId \\
tickTimerThreadId' \neq invalidThreadId \\
tickTimerThreadId' \neq idleThreadId' \\
allThreads' = \{\, idleThreadId', tickTimerThreadId' \,\} \\
allCondvars' = \{\, zThreadInstances(idleThreadId').builtinCondvarId, \\
\qquad zThreadInstances(tickTimerThreadId').builtinCondvarId \,\} \\
allTimers' = \{\, zThreadInstances(idleThreadId').delayTimerId, \\
\qquad zThreadInstances(tickTimerThreadId').delayTimerId \,\} \\
timerTicksSinceBoot' = 0 \\
interruptNestingLevelStack' = \theta\, InterruptNestingLevelStackInitialState \\
interruptNestingLevelStack'.zCpuId = cpuId \\
timerWheel' = \theta\, TimerWheelInitialState \\
zCpuInterruptMaskingState' = cpuInterruptsEnabled \\
zCpuPrivilege' = cpuUnprivileged \\
zMemoryProtectionState' = memoryProtectionOn \\
runnableThreadQueues'(min\ ValidInterruptPriorityIdType) = \langle idleThreadId \rangle \\
runnableThreadQueues'(max\ ValidInterruptPriorityIdType) = \langle tickTimerThreadId \rangle \\
\forall\, p : ValidThreadPrioirtyType \setminus \\
\qquad \{min\ ValidThreadPrioirtyType, \max\ ValidThreadPrioirtyType\} \bullet \\
\quad runnableThreadQueues'(p) = \emptyset
\end{array}
$$

$$
\begin{array}{l}
\textbf{InterruptNestingLevelStackInitialState} \\
\hline
InterruptNestingLevelStackType' \\
\hline
currentInterruptNestingCounter = 1
\end{array}
$$

InterruptNestingLevelInitialState

$InterruptNestingLevelType'$

$interruptId' = invalidInterruptId$

$interruptNestingCounter' = 0$

$savedStackPointer' = nullAddress$

$atomicLevel' = atomicLevelNones$

TimerWheelInitialState

$TimerWheelType'$

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

$currentWheelSpokeIndex = min\ ValidTimerWheelSpokeIndexType$

TimerTypeInitialState

ThreadTypeInitialState

CondvarTypeInitialState

MutexTypeInitialState

2.7 Starting the Per-CPU ***HiRTOS*** Thread Scheduler

When calling the `HiRTOS.Start_Thread_Scheduler` *HiRTOS* API, on a given CPU core, RTOS multi-tasking is started on the given CPU, as described by the precondition/postcondition contract shown below:

$$
\begin{array}{l}
\textbf{HiRtosStartThreadScheduler} \\
\hline
\Delta \mathit{HiRtosCpuInstanceType} \\
\hline
\mathit{threadSchedulerState} = \mathit{threadSchedulerStopped} \\
\mathit{currentThreadId} = \mathit{invalidThreadId} \\
\mathit{currentAtomicLevel} = \mathit{AtomicLevelNone} \\
\mathit{zCpuPrivilege} = \mathit{cpuPrivileged} \\
\mathit{currentCpuExecutionMode} = \mathit{cpuExecutingResetHandler} \\
\\
\mathit{threadSchedulerState}' = \mathit{threadSchedulerRunning} \\
\mathit{currentThreadId}' = \mathit{zGetHighestPriorityThread}(\mathit{cpuId}) \\
\mathit{currentAtomicLevel}' = \mathit{AtomicLevelNone} \\
\mathit{zCpuPrivilege} = \mathit{cpuUnprivileged} \\
\mathit{currentCpuExecutionMode}' = \mathit{cpuExecutingThread} \\
\hline
\end{array}
$$

2.8 Entering ***HiRTOS*** from Interrupt Context

After calling the `HiRTOS.Enter_Interrupt_Context` *HiRTOS* API, from an ISR on a given CPU core, RTOS multi-tasking the *HiRTOS* environment for interrupt context is entered.

$$
\begin{array}{l}
\textbf{HiRtosEnterInterruptContext} \\
\hline
\Delta \mathit{HiRtosInstanceType} \\
\mathit{zInterruptId}? : \mathit{InterruptIdType} \\
\hline
\hline
\end{array}
$$

2.9 Exiting ***HiRTOS*** from Interrupt Context

After calling the `HiRTOS.Exit_Interrupt_Context` *HiRTOS* API, from an ISR on a given CPU core, RTOS multi-tasking the *HiRTOS* environment for interrupt context is exited.

$$
\begin{array}{|l}
\hline
\textit{HiRtosExitInterruptContext} \\
\Delta \textit{HiRtosInstanceType} \\
\textit{zCpuId?} : \textit{CpuIdType} \\
\textit{zInterruptId?} : \textit{InterruptIdType} \\
\hline
\\
\hline
\end{array}
$$

Bibliography

[1] 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

[2] 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

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

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

[5] Mike Spivey, “The Fuzz checker”
http://spivey.oriel.ox.ac.uk/mike/fuzz

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

[7] 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

[8] AdaCore,“Formal Verification with GNATprove”
https://docs.adacore.com/spark2014-docs/html/ug/en/gnatprove.html