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

Germán Rivera
jgrivera67@gmail.com

December 30, 2022

Contents

1 Introduction **1**
- 1.1 Z Naming Conventions . . . 1
- 1.2 Major Design Decisions . . . 2

2 *HiRTOS* Z Specification **3**
- 2.1 *HiRTOS* Configuration Parameters . . . 3
- 2.2 *HiRTOS* Target Platform Parameters . . . 3
- 2.3 *HiRTOS* Primitive Types . . . 5
- 2.4 *HiRTOS* Axiomatic Definitions . . . 6
- 2.5 *HiRTOS* State Variables . . . 7
- 2.6 *HiRTOS* Initialization . . . 9
- 2.7 Starting the Per-CPU *HiRTOS* Thread Scheduler . . . 11
- 2.8 Entering *HiRTOS* from Interrupt Context . . . 11
- 2.9 Exiting *HiRTOS* from Interrupt Context . . . 11
  - 2.9.1 CPU Controllers . . . 13
  - 2.9.2 ExecutionContext . . . 16
  - 2.9.3 Threads . . . 16
  - 2.9.4 Interrupts . . . 17
  - 2.9.5 Timers . . . 17
  - 2.9.6 Mutexes . . . 17
  - 2.9.7 Condition Variables . . . 18
  - 2.9.8 Message Channels . . . 19
  - 2.9.9 Generic Data Structures . . . 20

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.

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*.

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

2.3 *HiRTOS* Primitive Types

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

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

2.4 *HiRTOS* Axiomatic Definitions

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

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

2.5 *HiRTOS* State Variables

$$
\begin{array}{l}
\textit{HiRtosType} \\
\hline
\textit{createdThreadInstances} : \textit{ValidThreadIdType} \rightarrowtail\!\!\!\!\!\!\shortmid\;\; \textit{ThreadType} \\
\textit{createdMutexInstances} : \textit{ValidMutexIdType} \rightarrowtail\!\!\!\!\!\!\shortmid\;\; \textit{MutexType} \\
\textit{createdCondvarInstances} : \textit{ValidCondvarIdType} \rightarrowtail\!\!\!\!\!\!\shortmid\;\; \textit{CondvarType} \\
\textit{createdTimerInstances} : \textit{ValidTimerIdType} \rightarrowtail\!\!\!\!\!\!\shortmid\;\; \textit{TimerType} \\
\textit{rtosCpuInstances} : \textit{ValidCpuIdType} \rightarrowtail\!\!\!\!\!\!\shortmid\;\; \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
\end{array}
$$

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

$$
\begin{array}{l}
\textit{HiRtosCpuInstanceType} \\
\hline
\textit{cpuId} : \textit{CpuIdType} \\
\textit{threadSchedulerState} : \textit{ThreadSchedulerStateType} \\
\textit{currentAtomicLevel} : \textit{AtomicLevelType} \\
\textit{currentCpuExecutionMode} : \textit{CpuExecutionModeType} \\
\textit{currentThreadId} : \textit{ThreadIdType} \\
\textit{timerTicksSinceBoot} : \textit{TimerTicksType} \\
\textit{idleThreadId} : \textit{ValidThreadIdType} \\
\textit{tickTimerThreadId} : \textit{ValidThreadIdType} \\
\textit{interruptNestingLevelStack} : \textit{InterruptNestingLevelStackType} \\
\textit{allThreads} : \textit{PerCpuThreadSetType} \\
\textit{allMutexes} : \textit{PerCpuMutexSetType} \\
\textit{allCondvars} : \textit{PerCpuCondvarSetType} \\
\textit{allTimers} : \textit{PerCpuTimerSetType} \\
\textit{runnableThreadQueues} : \textit{ValidThreadPriorityIdType} \rightarrowtail \textit{ThreadQueueType} \\
\textit{timerWheel} : \textit{TimerWheelType} \\
\textit{zCpuInterruptMaskingState} : \textit{CpuInterruptMaskingStateType} \\
\textit{zCpuPrivilege} : \textit{CpuPrivilegeType} \\
\textit{zMemoryProtectionState} : \textit{MemoryProtectionStateType} \\
\hline
\{\, \textit{idleThreadId}, \textit{tickTimerThreadId} \,\} \subseteq \textit{allThreads} \\[1ex]
\textit{tickTimerThreadId} \neq \textit{idleThreadId} \\[1ex]
\textit{threadSchedulerState} = \textit{threadSchedulerRunning} \Rightarrow \\
\quad \textit{currentThreadId} = \textit{zGetHighestPriorityThread}(\textit{cpuId}) \\[1ex]
\textit{zCpuInterruptMaskingState} = \textit{cpuInterruptsEnabled} \Leftrightarrow \\
\quad \textit{currentAtomicLevel} > \textit{AtomicLevelNoInterrupts} \\[1ex]
\textit{zCpuInterruptMaskingState} = \textit{cpuInterruptsDisabled} \Rightarrow \\
\quad \textit{zCpuPrivilege} = \textit{cpuPrivileged} \\[1ex]
\textit{currentAtomicLevel} < \textit{AtomicLevelNone} \Rightarrow \textit{zCpuPrivilege} = \textit{cpuPrivileged} \\[1ex]
\bigcap \{\, i : \textit{allThreads} \bullet \{\, \textit{zThreadInstances}(i).\textit{builtinCondvarId} \,\} \,\} = \emptyset \\[1ex]
\bigcap \{\, i : \textit{allThreads} \bullet \{\, \textit{zThreadInstances}(i).\textit{delayTimerId} \,\} \,\} = \emptyset \\[1ex]
\forall\, p : \textit{ValidThreadPrioirtyType} \bullet \\
\quad \forall\, t : \operatorname{ran} \textit{runnableThreadQueues}(p) \bullet t.\textit{priority} = p
\end{array}
$$

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:

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

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:

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

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:

HiRtosStartThreadScheduler

$\Delta HiRtosCpuInstanceType$

$threadSchedulerState = threadSchedulerStopped$

$currentThreadId = invalidThreadId$

$currentAtomicLevel = AtomicLevelNone$

$zCpuPrivilege = cpuPrivileged$

$currentCpuExecutionMode = cpuExecutingResetHandler$

$threadSchedulerState' = threadSchedulerRunning$

$currentThreadId' = zGetHighestPriorityThread(cpuId)$

$currentAtomicLevel' = AtomicLevelNone$

$zCpuPrivilege = cpuUnprivileged$

$currentCpuExecutionMode' = cpuExecutingThread$

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.

HiRtosEnterInterruptContext

$\Delta HiRtosInstanceType$

$zCpuId? : CpuIdType$

$zInterruptId? : InterruptIdType$

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.

$HiRtosExitInterruptContext$

$\Delta HiRtosInstanceType$
$zCpuId? : CpuIdType$
$zInterruptId? : InterruptIdType$

???

2.9.1 CPU Controllers

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

Invariants:

- There can be more than one interrupt with the same interrupt priority. Interrupt scheduling is done by hardware, by the interrupt controller.
- The same interrupt cannot be nested.

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

ThreadScheduler

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

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

Invariants:

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

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

$$\begin{array}{|l}\hline \textit{ThreadQueue} \\ \textit{GenericLinkedList}[\textit{Thread}] \\ \hline\end{array}$$

2.9.2 ExecutionContext

$$
\begin{array}{l}
\textit{ExecutionContext} \\
\hline
\mathit{cpuRegisters} : \mathit{CpuRegisterIdType} \rightarrowtail \mathit{CpuRegisterValueType} \\
\mathit{stackPointer} : \mathit{MemoryAddressType} \\
\mathit{cpuId} : \mathit{CpuIdType} \\
\mathit{cpuPrivilege} : \mathit{CpuPrivilegeType} \\
\mathit{contextType} : \mathit{ExecutionContextType} \\
\mathit{exeStackTopEnd} : \mathit{MemoryAddressType} \\
\mathit{exeStackBottomEnd} : \mathit{MemoryAddressType} \\
\hline
\mathit{stackPointer} \in \mathit{cpuRegisters} \\
\mathit{kWordValue}(\mathit{stackPointer}) \in \operatorname{dom} \mathit{executionStack} \\
\mathit{exeStackTopEnd} < \mathit{exeStackBottomEnd} \\
\mathit{exeStackTopEnd} \mathrel{..} \mathit{exeStackBottomEnd} \subset \mathit{kValidRamWordAddresses} \\
\operatorname{dom} \mathit{executionStack} = \mathit{exeStackTopEnd} + 1 \mathrel{..} \mathit{exeStackBottomEnd} \\
\operatorname{dom} \mathit{executionStack} \subset \mathit{kValidRamWordAddresses} \\
\operatorname{dom} \mathit{executionStack} \cap \mathit{kReadOnlyAddresses} = \emptyset \\
\hline
\end{array}
$$

2.9.3 Threads

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

User-created threads run in the CPU's unprivileged mode and system internal threads run in the CPU's privileged mode. This is to prevent user threads to execute privileged instructions. If a user thread needs to execute a provileged instruction, it needs to first switch the CPU to privileged mode.

Invariants:

- The current priority of a thread can never be lower than its base priority. The current priority can be higher than the base priority when it acquires a mutex that has higher priority than the thread's base priority.
- A thread never gets blocked trying to acquire a mutex that has the same priority as the thread. Still, the thread needs to acquire the mutex, since other threads with the same prioirity may also try to acquire the same mutex, if the running thread gets switched out due running out of its time slice.
- A thread should never try to acquire a mutex of lower priority than the thread's priority. Indeed, It does not need to, as it cannot be preemted by lower priority threads.

2.9.4 Interrupts

Interrupt

$$\begin{array}{l}
executionContext : ExecutionContext \\
interruptChannel : INTERRUPT_CHANNEL \\
isrFunction : kExecutableAddresses \\
\hline
executionContext.contextType = kInterruptContext \\
executionContext.cpuPrivilege = cpuPrivileged \\
\#executionContext.executionStack = kInterruptStackSizeInWords
\end{array}$$

Interrupt execution contexts run in privileged mode. To ensure that a higher priority interrupt is not delayed by a lower priority interrupt, nested interrupts are supported. To this end, interrupt service routines (ISRs) run with interrupts enabled by default. However, interrupts with the same or lower priority cannot interrupt the CPU until we finish servicing the current interrupt, as the interrupt controller is expected to only raise interrupts with higher priority than the current one being serviced. (The last step in servicing an interrupt is to notify the interrupt controller of the completion of servicing the interrupt).

2.9.5 Timers

Timer

$$counter : \mathbb{N}$$

2.9.6 Mutexes

Mutex
$waitingThreads : ThreadQueue$
$synchronizationScope : SynchronizationScopeType$
$priority : MutexPriorityType$

HiRTOS mutexes implement the priority ceiling protocol. That is, each mutex has a priority associated with it, which is the priority of the highest priority task that accesses the resource protected by the mutex, or the lowest interrupt priorirty, in case if an ISR accesses the resource protected by the mutex. The mutex is supposed be acquired by threads that have lower priority than the mutex's priority. If the mutex has prioirty higher or equal to the lowest interrupt priority, acquiring the mutex also disables interrupts in the CPU.

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

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

2.9.7 Condition Variables

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

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

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

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

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

2.9.8 Message Channels

MessageChannel

$GenericCircularBuffer[WORD\_LOCATION]$

2.9.9 Generic Data Structures

Generic Linked Lists

GenericLinkedList[ElementType]

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

$$listAnchor \notin zNodes$$

$$numNodes = \#zNodes$$

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

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

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

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

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

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

$$\#zElements = \#zNodes$$

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

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

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

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

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

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

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

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

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

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

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

Generic Circular Buffers

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

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

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