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

Germán Rivera
jgrivera67@gmail.com

August 4, 2022

Contents

Chapter 1

HiRTOS: A High Integrity RTOS

This document describes the design of *HiRTOS*, a high integrity real-time operating system kernel that supports multi-core systems. 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.

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 model-only variables that are not meant to be implemented in code.

1.2   Major Design Decisions

- ISRs are seen as hardware-scheduled threads that have higher priority than all software-scheduled threads. They always run in privileged mode and 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. Global variables in application code are read-only by default. To be able to write global variables, application code must request write permission to HiRTOS via a system call. MMIO space is not accessible by default. Application (or driver) code must request access (read-only or read-write permission) to HiRTOS via a system call.

1.3 HiRTOS Configuration Parameters

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

> $maxNumThreads : \mathbb{N}_1$
> $maxNumMutexes : \mathbb{N}_1$
> $maxNumCondvars : \mathbb{N}_1$
> $maxNumTimers : \mathbb{N}_1$
> $numThreadPriorities : \mathbb{N}_1$
> ───────────
> $maxNumThreads > 2$

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.

1.4 HiRTOS Target Platform Parameters

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

> $maxNumCpus : \mathbb{N}_1$
> $minMemoryAddress : \mathbb{N}$
> $maxMemoryAddress : \mathbb{N}_1$
> $numInterruptPriorities : \mathbb{N}_1$
> ───────────
> $minMemoryAddress < maxMemoryAddress$

1.5   HiRTOS Primitive Types

$CpuIdType == 0 .. maxNumCpus - 1$
$MemoryAddressType == minMemoryAddress .. maxMemoryAddress$
$ThreadIdType == 0 .. maxNumThreads$
$invalidThreadId == maxNumThreads$
$idleThreadId == 0$
$tickTimerThreadId == 1$
$ValidThreadIdType == ThreadType \setminus \{InvalidThreadId\}$
$ThreadPrioirtyType == 0 .. numThreadPriorities$
$invalidThreadPriority == numThreadPriorities$
$ValidThreadPriorityType == ThreadPriorityType \setminus \{InvalidThreadPriority\}$
$MutexIdType == 0 .. maxNumMutexes$
$invalidMutexId == maxNumMutexes$
$ValidMutexIdType == MutexType \setminus \{InvalidMutexId\}$
$CondvarIdType == 0 .. maxNumCondvars$
$invalidCondvarId == maxNumCondvars$
$ValidCondvarIdType == CondvarType \setminus \{InvalidCondvarId\}$
$TimerIdType == 0 .. maxNumTimers$
$invalidTimerId == maxNumTimers$
$ValidTimerIdType == TimerType \setminus \{InvalidTimerId\}$
$InterruptPrioirtyType == 0 .. numInterruptPriorities$
$invalidInterruptPriority == numInterruptPriorities$
$ValidInterruptPriorityIdType == InterruptPriorityType \setminus \{InvalidInterruptPriorityId\}$
$AtomicLevelType == 0 .. numInterruptPriorities + 1$
$atomicLevelNoInterrupts == min\ AtomicLevelType$
$atomicLevelSingleThread == max\ AtomicLevelType - 1$
$atomicLevelNone == max\ AtomicLevelType$
$CpuInterruptMaskingStateType ::= cpuInterruptsEnabled \mid cpuInterruptsDisabled$
$CpuPrivilegeType ::= cpuPrivileged \mid cpuUnprivileged$
$MemoryProtectionStateType ::= memoryProtectionOn \mid memoryProtectionOff$
$CpuExecutionModeType ::= cpuExecutingResetHandler \mid cpuExecutingInterruptHandler \mid c_{\ell}$
$ThreadStateType ::= threadNotCreated \mid threadRunnable \mid threadRunning \mid threadInterrupted$
$HiRtosStateType ::= threadSchedulerStopped \mid threadSchedulerRunning$
$TimerTicksType == \mathbb{N}$
$ThreadQueueType == \text{iseq } ValidThreadIdType$
$MutexListType == \text{iseq } ValidMutexIdType$
$MutexListType == \text{iseq } ValidMutexIdType$
$TimerListType == \text{iseq } ValidTimerIdType$

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

1.6 Per-CPU HiRTOS State Variables

There is a separate HiRTOS instance per CPU core in the target platform:

$$\begin{array}{l} hiRtosInstances : CpuIdType \rightarrowtail HiRtosInstanceType \\ \hline \forall\, cpuId : \mathrm{dom}\, hiRtosInstances \bullet hiRtosInstances(cpuId).cpuId = cpuId \end{array}$$

The state variables and internal data structures of each HiRTOS instance are modeled by the *HiRtosInstanceType* state space schema:

$$\begin{array}{l} \underline{HiRtosInstanceType} \\ cpuId : CpuIdType \\ threadSchedulerState : ThreadSchedulerStateType \\ currentAtomicLevel : AtomicLevelType \\ currentPrivilegeNestingCounter : PrivilegeNestingCounterType \\ currentThreadId : ThreadIdType \\ timerTicksSinceBoot : TimerTicksType \\ threadInstances : ValidThreadIdType \nrightarrow ThreadType \\ mutexInstances : ValidMutexIdType \nrightarrow MutexType \\ condvarInstances : ValidCondvarIdType \nrightarrow CondvarType \\ timerInstances : ValidTimerIdType \nrightarrow TimerType \\ interruptNestingLevelStack : InterruptNestingLevelStackType \\ runnableThreadQueues : ValidThreadPriorityIdType \rightarrowtail ThreadQueueType \\ timerWheel : TimerWheelType \\ zCpuInterruptMaskingState : CpuInterruptMaskingStateType \\ zCpuPrivilege : CpuPrivilegeType \\ zCpuExecutionMode : CpuExecutionModeType \\ zMemoryProtectionState : MemoryProtectionStateType \\ \hline threadSchedulerState = ThreadSchedulerRunning \Rightarrow currentThreadId \in \mathrm{dom}\, threadInstances \\ \\ zCpuInterruptMaskingState = cpuInterruptsEnabled \Leftrightarrow currentAtomicLevel > AtomicLevelNoInterr \\ \\ zCpuInterruptMaskingState = cpuInterruptsDisabled \Rightarrow zCpuPrivilege = cpuPrivileged \\ \\ currentAtomicLevel < AtomicLevelNone \Rightarrow zCpuPrivilege = cpuPrivileged \\ \\ currentPrivilegeNestingCounter > 0 \Leftrightarrow zCpuPrivilege = cpuPrivileged \\ \\ zCpuExectionMode \neq cpuExecutingThread \Rightarrow zCpuPrivilege = cpuPrivileged \end{array}$$

1.6.1 Per-CPU HiRTOS Initialization

After calling the `HiRTOS.Initialize` HiRTOS API, on a given CPU core, the state of the corresponding HiRTOS instance is as described below:

HiRtosInitialize

$HiRtosInstanceType'$
$zCpuId? : CpuIdType$

$cpuId' = zCpuId?$

$threadSchedulerState' = threadSchedulerStopped$

$currentAtomicLevel' = AtomicLevelNone$

$currentPrivilegeNestingCounter' = 1$

$currentThreadId' = InvalidThreadId$

$timerTicksSinceBoot' = 0$

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

$mutexInstances' = \emptyset$

$condvarInstances' = \emptyset$

$timerInstances' = \emptyset$

$\theta\,interruptNestingLevelStack' = \theta\,InitInterruptNestingLevelStack$

$\theta\,timerWheel' = \theta\,InitTimerWheel$

$zCpuInterruptMaskingState' = cpuInterruptsEnabled$

$zCpuPrivilege' = cpuPrivileged$

$zMemoryProtectionState' = memoryProtectionOn$

$zCpuExecutionMode' = cpuExecutingResetHandler$

$runnableThreadQueues'(min\ ValidInterruptPriorityIdType) = \langle idleThreadId \rangle$

$runnableThreadQueues'(max\ ValidInterruptPriorityIdType) = \langle tickTimerThreadId \rangle$

$\forall\, p : ValidThreadPrioirtyType \setminus \{min\ ValidThreadPrioirtyType, max\ ValidThreadPrioirty$
    $runnableThreadQueues'(p) = \emptyset$

1.6.2 Starting the Per-CPU HiRTOS Thread Scheduler

After calling the `HiRTOS.Start_Thread_Scheduler` HiRTOS API, on a given CPU core, RTOS multi-tasking is started on the given CPU, as described below:

HiRtosStartThreadScheduler

$\Delta HiRtosInstanceType$
$zCpuId? : CpuIdType$

1.6.3 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 ──────────────────────────
│ ΔHiRtosInstanceType
│ zCpuId? : CpuIdType
│ zInterruptId? : InterruptIdType
│ ┌────────────
│ └────────────
└────────────────────────────────────────────────────────
```

1.6.4 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 ──────────────────────────
│ ΔHiRtosInstanceType
│ zCpuId? : CpuIdType
│ zInterruptId? : InterruptIdType
│ ┌────────────
│ └────────────
└───────────────────────────────────────────────────────
```

???

1.6.5 CPU Controllers

CpuController

ThreadScheduler
cpuId : *CpuIdType*
zExecutionContexts : $\mathbb{F}_1$ *ExecutionContext*
preemptedBy : *ExecutionContext* $\rightarrowtail$ *ExecutionContext*
timers : $\mathbb{F}$ *Timer*
zIterruptChannelToInterrupt : *INTERRUPT_CHANNEL* $\rightarrowtail$ *Interrupt*
interrupts : $\mathbb{F}_1$ *Interrupt*
tickTimerInterrupt : *Interrupt*
runningExecutionContext : *ExecutionContext*
nestedInterruptCount : $0 \ldots kMaxNumInterruptChannelsPerCpu$
activeInterruptsBitMap : $\mathbb{F}$ *INTERRUPT_CHANNEL*
activeInterrupts : $\mathbb{F}$ *Interrupt*

---

ran *zIterruptChannelToInterrupt* = *interrupts*

zExecutionContexts =
    $\{t : threads \bullet t.executionContext\} \cup \{i : interrupts \bullet i.executionContext\}$

$\{t : threads \bullet t.executionContext\} \cap \{i : interrupts \bullet i.executionContext\} = \emptyset$

$\forall et : zExecutionContexts \bullet et.cpuId = cpuId$

activeInterrupts = $\{iv : activeInterruptsBitMap \bullet zIterruptChannelToInterrupt(iv)\}$

nestedInterruptCount = #*activeInterrupts*

nestedInterruptCount = $0 \Leftrightarrow$
    runningExecutionContext $\in \{t : zRunnableThreads \bullet t.executionContext\}$

nestedInterruptCount > $0 \Leftrightarrow$
    runningExecutionContext $\in \{i : activeInterrupts \bullet i.executionContext\}$

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 ──────────────────────

$zThreadIdToThread : THREAD\_ID \rightarrowtail Thread$
$threads : \mathbb{F}_1\ Thread$
$zUserThreads : \mathbb{F}\ Thread$
$zSystemThreads : \mathbb{F}_1\ Thread$
$idleThread : Thread$
$runningThread : Thread$
$runnableThreadPrioritiesBitMap : \mathbb{F}_1\ THREAD\_PRIO$
$runnableThreadQueues : THREAD\_PRIO \rightarrowtail ThreadQueue$
$zRunnableThreads : \mathbb{F}_1\ Thread$

───────────────────────────────────

$\mathrm{ran}\ zThreadIdToThread = threads$

$zRunnableThreads =$
$\quad \bigcup \{i : THREAD\_PRIO \bullet \mathrm{ran}(runnableThreadQueues(i)).zElements\}$

$zRunnableThreads \neq \emptyset \wedge zRunnableThreads \subseteq threads$

$threads = zUserThreads \cup zSystemThreads$

$zUserThreads \cap zSystemThreads = \emptyset$

$\forall\, t : zSystemThreads \bullet t.executionContext.cpuPrivilege = cpuPrivileged$

$\forall\, t : zUserThreads \bullet$
$\quad t.executionContext.contextType = threadContext$

$idleThread \in zSystemThreads$

$zThreadIdToThread(0) = idleThread$

$runningThread \in zRunnableThreads \wedge runningThread.state = kRunning$

$\forall\, t : zRunnableThreads \setminus \{runningThread\} \bullet t.state = kRunnable$

$\forall\, t : threads \setminus zRunnableThreads \bullet$
$\quad t.state \notin \{kRunnable, kRunning\}$

$\mathrm{ran}(runnableThreadQueues(kLowestThreadPriority)).zElements = \{idleThread\}$

$\forall\, t : threads \bullet$
$\quad runningThread.currentPriority \geq t.currentPriority$

$\forall\, prio : runnableThreadPrioritiesBitMap \bullet prio \in \mathrm{dom}\ runnableThreadQueues$

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.

ThreadQueue
GenericLinkedList[*Thread*]

1.6.6   ExecutionContext

ExecutionContext _____

$cpuRegisters : CpuRegisterIdType \rightarrowtail CpuRegisterValueType$
$stackPointer : MemoryAddressType$
$cpuId : CpuIdType$
$cpuPrivilege : CpuPrivilegeType$
$contextType : ExecutionContextType$
$exeStackTopEnd : MemoryAddressType$
$exeStackBottomEnd : MemoryAddressType$

_____

$stackPointer \in cpuRegisters$

$kWordValue(stackPointer) \in \text{dom } executionStack$

$exeStackTopEnd < exeStackBottomEnd$

$exeStackTopEnd \mathinner{\ldotp\ldotp} exeStackBottomEnd \subset kValidRamWordAddresses$

$\text{dom } executionStack = exeStackTopEnd + 1 \mathinner{\ldotp\ldotp} exeStackBottomEnd$

$\text{dom } executionStack \subset kValidRamWordAddresses$

$\text{dom } executionStack \cap kReadOnlyAddresses = \emptyset$

1.6.7   Threads

Thread _____

$executionContext : ExecutionContext$
$threadID : THREAD\_ID$
$threadFunction : kExecutableAddresses$
$state : THREAD\_STATE$
$basePriority : THREAD\_PRIO$
$currentPriority : THREAD\_PRIO$
$listNode : LIST\_NODE$
$deadlineToRun : \mathbb{N}$

_____

$currentPriority \geq basePriority$

$executionContext.contextType = kThreadContext$

$\#executionContext.executionStack = kThreadStackSizeInWords$

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.

1.6.8 Interrupts

Interrupt
executionContext : *ExecutionContext*
interruptChannel : *INTERRUPT_CHANNEL*
isrFunction : *kExecutableAddresses*

executionContext.contextType = *kInterruptContext*

executionContext.cpuPrivilege = *cpuPrivileged*

#*executionContext.executionStack* = *kInterruptStackSizeInWords*

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).

1.6.9 Timers

Timer
counter : $\mathbb{N}$

1.6.10 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.

1.6.11   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.

1.6.12   Message Channels

```
┌─ MessageChannel ──────────────────────────────────────────────
│  GenericCircularBuffer[WORD_LOCATION]
└────────────────────────────────────────────────────────────────
```

1.6.13   Generic Data Structures

Generic Linked Lists

$\_\_$ *GenericLinkedList*[*ElementType*] $_____$
listAnchor : *LIST_NODE*
numNodes : $\mathbb{N}$
zNodes : $\mathbb{F}$ *LIST_NODE*
zElements : iseq *ElementType*
zNodeToElem : *LIST_NODE* $\rightarrowtail\!\!\!\to$ *ElementType*
zNextNode : *LIST_NODE* $\rightarrowtail\!\!\!\to$ *LIST_NODE*
zPrevNode : *LIST_NODE* $\rightarrowtail\!\!\!\to$ *LIST_NODE*
zNodeToListAnchor : *LIST_NODE* $\rightarrowtail\!\!\!\to$ *LIST_NODE*
$_____$

listAnchor $\notin$ *zNodes*

numNodes = #*zNodes*

dom *zNodeToElem* = *zNodes*

ran *zNodeToElem* = ran *zElements*

dom *zNextNode* = *zNodes* $\cup$ {*listAnchor*}

ran *zNextNode* = *zNodes* $\cup$ {*listAnchor*}

dom *zPrevNode* = dom *zNextNode*

ran *zPrevNode* = 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

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

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

$\forall\, x : zNodes;\ k : 1 \mathinner{\ldotp\ldotp} \#zNodes \bullet$
    $zNextNode^{k}(x) \neq x \land zPrevNode^{k}(x) \neq x$

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

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

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

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

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

Generic Circular Buffers

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

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`