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

Germán Rivera
jgrivera67@gmail.com

October 11, 2021

Contents

1 HiRTOS: A High Integrity RTOS **1**
1.1 Z Naming Conventions 1
1.2 Numeric Constants 2
1.3 Primitive Types 3
1.4 Structural Constants 3
1.5 State Variables 3
1.5.1 Execution Controller 6
1.5.2 CPU Controllers 8
1.5.3 ExecutionContext 11
1.5.4 Threads 11
1.5.5 Interrupts 12
1.5.6 Timers 12
1.5.7 Mutexes 12
1.5.8 Condition Variables 13
1.5.9 Message Channels 14
1.5.10 Generic Data Structures 15

Chapter 1

HiRTOS: A High Integrity RTOS

This document describes the design of *HiRTOS*, a high integrity real-time operating system kernel. The design is presented using the Z notation [1, 2]. 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 [3], 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 *ghost* prefix are model-only variables that are not meant to be implemented in code.

1.2 Numeric Constants

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

$$
\begin{array}{|l}
\mathit{maxNumCpus} : \mathbb{N}_1 \\
\mathit{numCpuRegisters} : \mathbb{N}_1 \\
\mathit{cpuRegisterWidth} : \mathbb{N}_1 \\
\mathit{minMemoryAddress} : \mathbb{N} \\
\mathit{maxMemoryAddress} : \mathbb{N}_1 \\
\mathit{maxNumThreads} : \mathbb{N}_1 \\
\mathit{maxNumMutexes} : \mathbb{N}_1 \\
\mathit{maxNumCondvars} : \mathbb{N}_1 \\
\mathit{maxNumInterrupts} : \mathbb{N}_1 \\
\mathit{numThreadPriorities} : \mathbb{N}_1 \\
\mathit{numInterruptPriorities} : \mathbb{N}_1 \\
\mathit{lowestThreadPriority} : \mathbb{N}_1 \\
\mathit{highestThreadPriority} : \mathbb{N}_1 \\
\mathit{lowestInterruptPriority} : \mathbb{N}_1 \\
\mathit{highestInterruptPriority} : \mathbb{N}_1 \\
\hline
\mathit{minMemoryAddress} < \mathit{maxMemoryAddress} \\
\mathit{highestInterruptPriority} = 1 \\
\mathit{lowestInterruptPriority} = \mathit{numInterruptPriorities} \\
\mathit{highestThreadPriority} = \mathit{lowestInterruptPriority} + 1 \\
\mathit{lowestThreadPriority} = \mathit{lowestInterruptPriority} + \mathit{numThreadPriorities}
\end{array}
$$

1.3 Primitive Types

$$\begin{array}{l}
\mathit{CpuIdType} == 1 \mathrel{..} \mathit{maxNumCpus} \\
\mathit{CpuRegisterIdType} == 1 \mathrel{..} \mathit{numCpuRegisters} \\
\mathit{CpuRegisterValueType} == 0 \mathrel{..} 2^{\mathit{cpuRegisterWidth}} - 1 \\
\mathit{MemoryAddressType} == \mathit{minMemoryAddress} \mathrel{..} \mathit{maxMemoryAddress} \\
\mathit{ThreadIdType} == 1 \mathrel{..} \mathit{maxNumThreads} \\
\mathit{MutexIdType} == 1 \mathrel{..} \mathit{maxNumMutexes} \\
\mathit{CondvarIdType} == 1 \mathrel{..} \mathit{maxNumCondvars} \\
\mathit{InterruptIdType} == 1 \mathrel{..} \mathit{maxNumInterrupts} \\
\mathit{InterruptPrioirtyType} == 1 \mathrel{..} \mathit{numInterruptPriorities} \\
\mathit{ThreadPrioirtyType} == \\
\qquad \mathit{numInterruptPriorities} + 1 \mathrel{..} \mathit{numInterruptPriorities} + \mathit{numThreadPriorities} \\
\mathit{MutexPriorityType} == \mathit{InterruptPrioirtyType} \cup \mathit{ThreadPrioirtyType} \\
\mathit{CpuInterruptsStateType} ::= \mathit{cpuInterruptsEnabled} \mid \mathit{cpuInterruptsDisabled} \\
\mathit{CpuPrivilegeType} ::= \mathit{cpuPrivileged} \mid \mathit{cpuUnprivileged} \\
\mathit{ExecutionContextType} ::= \mathit{interruptContext} \mid \mathit{threadContext} \\
\mathit{ThreadStateType} ::= \mathit{threadNotCreated} \mid \mathit{threadRunnable} \mid \mathit{threadRunning} \mid \mathit{threadBlocked} \\
\mathit{SynchronizationScopeType} ::= \mathit{localCpuOnly} \mid \mathit{interCpu}
\end{array}$$

For both threads and interrupts, lower priority numbers represent higher priorities. Interrupts have higher priority than threads. That is, the lowest priority interrupt has higher priority than the highest priority thread.

1.4 Structural Constants

$$\begin{array}{|l}
\mathit{threadsMap} : \mathit{ThreadIdType} \rightarrowtail \mathit{Thread} \\
\mathit{mutexesMap} : \mathit{MutexIdType} \rightarrowtail \mathit{Mutex} \\
\mathit{condvarsMap} : \mathit{CondvarIdType} \rightarrowtail \mathit{Condvar} \\
\mathit{interruptsMap} : \mathit{InterruptIdType} \rightarrowtail \mathit{Interrupt}
\end{array}$$

1.5 State Variables

$$
\begin{array}{l}
\textit{HiRTOS} \\
\hline
\textit{ExecutionController} \\
\textit{mutexes} : \mathbb{F}\,\textit{Mutex} \\
\textit{condvars} : \mathbb{F}\,\textit{Condvar} \\
\textit{messageChannels} : \mathbb{F}\,\textit{MessageChannel} \\
\textit{zMutexOwner} : \textit{Mutex} \nrightarrow \textit{ExecutionContext} \\
\textit{zMutexWaiters} : \textit{Thread} \nrightarrow \textit{Mutex} \\
\textit{zCondvarWaiters} : \textit{Thread} \nrightarrow \textit{Condvar} \\
\textit{zCondvarToMutex} : \textit{Condvar} \nrightarrow \textit{Mutex} \\
\hline
\operatorname{dom} \textit{zMutexOwner} \subseteq \textit{mutexes} \\[1ex]
\operatorname{ran} \textit{zMutexOwner} \subset \\
\qquad \{t : \textit{zThreads} \bullet t.\textit{executionContext}\} \cup \{i : \textit{zInterrupts} \bullet i.\textit{executionContext}\} \\[1ex]
\operatorname{dom} \textit{zMutexWaiters} \subset \textit{zThreads} \land \operatorname{ran} \textit{zMutexWaiters} \subseteq \textit{mutexes} \\[1ex]
\operatorname{dom} \textit{zCondvarWaiters} \subset \textit{zThreads} \land \operatorname{ran} \textit{zCondvarWaiters} \subseteq \textit{condvars} \\[1ex]
\operatorname{dom} \textit{zMutexWaiters} \cap \operatorname{dom} \textit{zCondvarWaiters} = \emptyset \\[1ex]
\#\textit{zMutexWaiters} < \#\textit{zThreads} \land \#\textit{zCondvarWaiters} < \#\textit{zThreads} \\[1ex]
\forall\, \textit{cpu} : \textit{CpuIdType} \bullet \\
\qquad (\textit{zCpuIdToCpuController}(\textit{cpu})).\textit{zRunnableThreads} \cap \\
\qquad (\operatorname{dom} \textit{zMutexWaiters} \cup \operatorname{dom} \textit{zCondvarWaiters}) = \emptyset \\[1ex]
\forall\, t : \textit{zThreads} \bullet \\
\qquad t.\textit{executionContext} \notin \operatorname{ran} \textit{zMutexOwner} \Rightarrow \\
\qquad\qquad t.\textit{currentPriority} = t.\textit{basePriority} \\[1ex]
\{\textit{mq} : \textit{messageChannels} \bullet \textit{mq}.\textit{mutex}\} \subset \textit{mutexes} \\[1ex]
\{\textit{mq} : \textit{messageChannels} \bullet \textit{mq}.\textit{notEmptyCondvar}\} \cup \\
\{\textit{mq} : \textit{messageChannels} \bullet \textit{mq}.\textit{notFullCondvar}\} \subset \textit{condvars} \\[1ex]
\forall\, \textit{cv} : \operatorname{dom} \textit{zCondvarToMutex} \bullet \\
\qquad \textit{cv}.\textit{synchronizationScope} \neq \textit{kLocalCpuInterruptAndThread} \land \\
\qquad \textit{cv}.\textit{synchronizationScope} = (\textit{zCondvarToMutex}(\textit{cv})).\textit{synchronizationScope} \\
\hline
\end{array}
$$

Invariants:

- The same thread cannot be waiting for more than one mutex.
- The same mutex cannot be owned by more than one thread.
- The same thread cannot be waiting on more than one condition variable.
- The same thread cannot be waiting on a mutex and a condition variable at the same time.

- The same thread cannot be both runnable and blocked on a condvar or mutex.
- ISRs can never wait on mutexes or condvars. However, ISRs can signal condvars for which waiting threads call “wait for interrupt”.
- The current priority of a thread that does not own a mutex must always be its base priority.

1.5.1 Execution Controller

$$
\begin{array}{|l}
\hline
\mathit{ExecutionController} \\
\mathit{zCpuIdToCpuController} : \mathit{CpuIdType} \rightarrowtail \mathit{CpuController} \\
\mathit{cpuControllers} : \mathbb{F}_1\ \mathit{CpuController} \\
\mathit{zExecutionContexts} : \mathbb{F}_1\ \mathit{ExecutionContext} \\
\mathit{zThreads} : \mathbb{F}\ \mathit{Thread} \\
\mathit{zInterrupts} : \mathbb{F}_1\ \mathit{Interrupt} \\
\mathit{zExecutionContextToCpu} : \mathit{ExecutionContext} \mathrel{+\!\!\!\!\!\twoheadrightarrow} \mathit{CpuIdType} \\
\mathit{zThreadToExecutionContext} : \mathit{Thread} \mathrel{+\!\!\!\!\!\rightarrowtail} \mathit{ExecutionContext} \\
\mathit{zInterruptToExecutionContext} : \mathit{Interrupt} \mathrel{+\!\!\!\!\!\rightarrowtail} \mathit{ExecutionContext} \\
\hline
\operatorname{ran} \mathit{zCpuIdToCpuController} = \mathit{cpuControllers} \\
\forall \mathit{cpu} : \mathit{CpuIdType} \bullet (\mathit{zCpuIdToCpuController}(\mathit{cpu})).\mathit{cpuId} = \mathit{cpu} \\
\bigcap\{\mathit{cpuC} : \mathit{cpuControllers} \bullet \mathit{cpuC}.\mathit{threads}\} = \emptyset \\
\bigcup\{\mathit{cpuC} : \mathit{cpuControllers} \bullet \mathit{cpuC}.\mathit{threads}\} = \mathit{zThreads} \\
\bigcap\{\mathit{cpuC} : \mathit{cpuControllers} \bullet \mathit{cpuC}.\mathit{interrupts}\} = \emptyset \\
\bigcup\{\mathit{cpuC} : \mathit{cpuControllers} \bullet \mathit{cpuC}.\mathit{interrupts}\} = \mathit{zInterrupts} \\
\mathit{zExecutionContexts} = \\
\qquad \{t : \mathit{zThreads} \bullet t.\mathit{executionContext}\} \cup \{i : \mathit{zInterrupts} \bullet i.\mathit{executionContext}\} \\
\{t : \mathit{zThreads} \bullet t.\mathit{executionContext}\} \cap \{i : \mathit{zInterrupts} \bullet i.\mathit{executionContext}\} = \emptyset \\
\operatorname{dom} \mathit{zExecutionContextToCpu} = \mathit{zExecutionContexts} \\
\operatorname{dom} \mathit{zThreadToExecutionContext} = \mathit{zThreads} \\
\operatorname{ran} \mathit{zThreadToExecutionContext} = \{t : \mathit{zThreads} \bullet t.\mathit{executionContext}\} \\
\operatorname{dom} \mathit{zInterruptToExecutionContext} = \mathit{zInterrupts} \\
\operatorname{ran} \mathit{zInterruptToExecutionContext} = \{i : \mathit{zInterrupts} \bullet i.\mathit{executionContext}\} \\
\forall \mathit{et} : \mathit{zExecutionContexts} \bullet \mathit{zExecutionContextToCpu}(\mathit{et}) = \mathit{et}.\mathit{cpuId} \\
\bigcap\{\mathit{et} : \mathit{zExecutionContexts} \bullet \mathit{et}.\mathit{executionStack}\} = \emptyset \\
\hline
\end{array}
$$

An execution context can correspond to a software-scheduled thread or to an interrupt. Interrupts are seen as “hardware-scheduled” threads that have higher priority than software-scheduled threads. Thus, an interrupt can be seen as a hardware thread that can preempt the highest priority software thread.

Invariants:

- Every thread must be assigned to a CPU. This is done at thread creation time and the thread cannot be moved to another CPU.

- Every interrupt source must be assigned to a CPU and must always be serviced by that CPU.
- The same thread cannot be assigned to more than one CPU.
- The same interrupt source cannot be assigned to more than one CPU.
- Every execution context is assigned to a fixed CPU. The same execution context cannot be assigned to two different CPUs. Every CPU has at least one execution context (if nothing else, its idle thread).

1.5.2 CPU Controllers

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

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

$$
\begin{array}{l}
\mathit{zThreadIdToThread} : \mathit{THREAD\_ID} \pinj \mathit{Thread} \\
\mathit{threads} : \mathbb{F}_1\ \mathit{Thread} \\
\mathit{zUserThreads} : \mathbb{F}\ \mathit{Thread} \\
\mathit{zSystemThreads} : \mathbb{F}_1\ \mathit{Thread} \\
\mathit{idleThread} : \mathit{Thread} \\
\mathit{runningThread} : \mathit{Thread} \\
\mathit{runnableThreadPrioritiesBitMap} : \mathbb{F}_1\ \mathit{THREAD\_PRIO} \\
\mathit{runnableThreadQueues} : \mathit{THREAD\_PRIO} \rightarrowtail \mathit{ThreadQueue} \\
\mathit{zRunnableThreads} : \mathbb{F}_1\ \mathit{Thread}
\end{array}
$$

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

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.

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

1.5.3 ExecutionContext

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

1.5.4 Threads

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

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.

1.5.5 Interrupts

$$\begin{array}{|l}
\multicolumn{1}{l}{\textit{Interrupt}} \\
\hline
\textit{executionContext} : \textit{ExecutionContext} \\
\textit{interruptChannel} : \textit{INTERRUPT\_CHANNEL} \\
\textit{isrFunction} : \textit{kExecutableAddresses} \\
\hline
\textit{executionContext.contextType} = \textit{kInterruptContext} \\
\textit{executionContext.cpuPrivilege} = \textit{cpuPrivileged} \\
\#\textit{executionContext.executionStack} = \textit{kInterruptStackSizeInWords} \\
\hline
\end{array}$$

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).

1.5.6 Timers

$$\begin{array}{|l}
\multicolumn{1}{l}{\textit{Timer}} \\
\hline
\textit{counter} : \mathbb{N} \\
\hline
\end{array}$$

1.5.7 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.

1.5.8 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.

1.5.9 Message Channels

MessageChannel

$$GenericCircularBuffer[WORD\_LOCATION]$$

1.5.10 Generic Data Structures

Generic Linked Lists

$$GenericLinkedList[ElementType]$$

$$listAnchor : LIST\_NODE$$

$$numNodes : \mathbb{N}$$

$$zNodes : \mathbb{F}\, LIST\_NODE$$

$$zElements : \operatorname{iseq} ElementType$$

$$zNodeToElem : LIST\_NODE \pinj ElementType$$

$$zNextNode : LIST\_NODE \pinj LIST\_NODE$$

$$zPrevNode : LIST\_NODE \pinj LIST\_NODE$$

$$zNodeToListAnchor : LIST\_NODE \pinj LIST\_NODE$$

$$listAnchor \notin zNodes$$

$$numNodes = \#zNodes$$

$$\operatorname{dom} zNodeToElem = zNodes$$

$$\operatorname{ran} zNodeToElem = \operatorname{ran} zElements$$

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

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

$$\operatorname{dom} zPrevNode = \operatorname{dom} zNextNode$$

$$\operatorname{ran} zPrevNode = \operatorname{ran} zNextNode$$

$$\#zElements = \#zNodes$$

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

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

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

$$\begin{aligned}&\forall x : zNodes \bullet\\ &\quad zPrevNode(zNextNode(x)) = x \land zNextNode(zPrevNode(x)) = x \land\\ &\quad zNodeToListAnchor(x) = listAnchor\end{aligned}$$

$$\begin{aligned}&\forall x : zNodes \bullet\\ &\quad zNextNode^{\#zNodes+1}(x) = x \land zPrevNode^{\#zNodes+1}(x) = x\end{aligned}$$

$$\begin{aligned}&\forall x : zNodes;\ k : 1 \mathrel{..} \#zNodes \bullet\\ &\quad zNextNode^{k}(x) \neq x \land zPrevNode^{k}(x) \neq x\end{aligned}$$

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

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

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

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

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

Generic Circular Buffers

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

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

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

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

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