Questions and Answers on RTSJ

Greg Bollella
Real-Time Java Expert Group lead, IBM
May 2000

Many of you have reviewed the *Real-time Specification for Java (RTSJ)* and provided lots of great feedback. Greg Bollella, a member of the Real-time for Java Experts Group, has responded to your questions, comments, and requests. Read on to find his position on the numerous issues that were raised by our real-time Java users.

Contents:

The questions and comments have been arranged according to the sections of the spec that they pertain to.

Overview

Question: "The ATC-deferred sections are synchronized code, constructors, and finally clauses." How is a finally clause identified by the virtual machine? There is no special construct for a finally clause on the bytecode level.

> **Bollella:** We changed finally clauses to be interruptible. (We've been considering doing this for awhile and your comment pushed us. Thanks.)

Question: The method modifiers `synchronized` and `native` define implementation characteristics, not general semantics. They should not appear in the header for any method in a specification unless the intent is to mandate the way in which those methods are implemented.

> **Bollella:** We agree that `native` should not appear. But `synchronized` must. Use of `synchronized` has exact semantics with respect to access to the code within the synchronized block.

Question: Point 7, 1.3.6 seems to be in conflict with normal Java. A thread would normally have the InterruptedException raised if t.interrupt() is called.

> **Bollella:** We fixed point 7. Text from the current spec is as follows:
>
> The RTSJ will provide mechanisms and programming disciplines to allow applications to bound waiting on I/O calls. There are two cases: (1) the device on which I/O is being performed (and thus its associated stream) is no longer needed and (2) timed, non-blocking I/O (where the device and associated streams remain viable). For case 1 the RTSJ requires that when stream.close() is called on a stream all blocked I/O calls will throw appropriate instances of IOException. Note that this requirement adds additional semantics to stream.close() which require blocked calls to throw an appropriate

Abbreviations

- **AE** -- The `AsyncEvent` class, an instance of the class, or a generic asynchronous event
- **AEH** -- The `AsynchEventHandler` class, an instance of the class, or the logic that will execute
- **AI** -- Asynchronously interruptible; means the referenced logic is interruptible
- **AIE** -- The `AsynchronouslyInterruptedException` class, an instance of the class, or the RTSJ mechanism
- **ASE** -- User shorthand for `AsyncEventHandler`
- **ATC** -- Asynchronous transfer of control; means the RTSJ mechanism that allows programatic initiation of a control transfer
- **EDF** -- Earliest deadline first; a type of scheduling discipline
- **GC** -- Garbage collector
- **RI** -- Reference implementaion
- **RT** -- Real-time

exception in addition to just checking for closed streams at the commencement of the I/O call. For case 2 the RTSJ recommends a programming discipline in which one thread uses the blocking calls from java.io.* and provides timed, non-blocking methods used by other threads.

- **RTSJ** -- The Real-Time Specification for Java
- **RTJ** -- Real-time Java

Threads

Question: You currently allow scheduling parameters on both RealtimeThreads and handlers. It seems possible to write code according to the standard that has a contradiction in itself: for example, write a periodic EventHandler that is mapped to a periodic RealtimeThread that has a completely different period. What would that mean?

Bollella: The threads to which AEHs are mapped are not visible to the application. They assume the parameters of the AEH.

Question: Regarding the interaction of RT and non-RT threads via locked shared objects: while at the application level, I can see how the waitFreeQueues would be used, I can't see how the implicit sharing that occurs within the core classes can be avoided. From class loading to string interning, RT and non-RT threads are inevitably going to be sharing objects using synchronized access -- and thus a potential for GC interruption occurs in the RT thread. Even thread creation itself will use the lock of the common ThreadGroup object.

This could be considered a quality-of-implementation issue but I think it warrants discussion, if not resolution, within the specification.

Bollella: For initialization, class loading, etc. there will be synchronization points between RT and non-RT threads. This is OK. Application logic should not use explicit synchronization between non-RT and RT or NoHeapRT threads if GC interruption is not desired. Note that careful documentation of java.* packages is required of implementation vendors if application logic is expected to use some of the classes and methods without GC interruption.

Scheduling

Question: Some of the wording of the standard outlaws EDF in our eyes. Here are examples (page numbers from 0.9 document):

Page 18,line 5: "By fixed-priority we mean that the system does not change the priority of any RealtimeThread or NoHeapRealtimeThread." EDF dynamically reassigns priorities all the time, therefore the sentence contradicts its use completely. (It does not even work if we say that we map all possible deadlines to millions(!) of priorities: then a periodic thread still gets a new "priority" at each invocation.)

Page 23, Point 2 in 2.1: "Any scheduling policy in an implementation other than the default scheduling policy must be available to instances of RealtimeThread and NoHeapRealtimeThread."

Bollella: We've added wording to be more clear. What we mean when we talk about priority is for threads scheduled by the required base scheduler.

Question: There seems to be no mention of interrupts and how they map to AsyncEvents.

Bollella: By design. All the RTSJ says is that the implementation must provide an implementation-dependent mechanism for binding AsyncEvents to happenings.

Question: If we first look at a non-EDF implementation of the standard, you seem to have five distinct types of threads or handlers:

- *1* Thread
- *2* RealtimeThread (but not an EventHandler)
- *3* NoHeapRealtimeThread (but not an event handler)
- *4* EventHandler that can be mapped as RealtimeThread
- *5* EventHandler that can be mapped as NoHeapRealtimeThread

You (deliberately?) leave it unspecified, between which of the five types synchronization (by any means) is legal. (You only make it clear, that it is illegal between *2* and *3*.) We think that the standard is incomplete and implementations will be difficult without this specification.

For example, can I synchronize between *1* and *2*? If I can, a regular Java Thread could suddenly run at priority 28 (because of priority inversion avoidance). Is that legal?

Even more important: can you synchronize between *2* and *4*? Or between *3* and *5*? Don't say "yes" lightly! This has severe implications on EDF and other non-priority based implementations!

Now let's look at an EDF implementation of the standard: the document 0.9 seems to imply that of the five types of threads and handlers, four can be scheduled as EDF, namely *2* to *5*. (Please confirm this view). In other words: you would now have nine types of threads/handlers:

- *1* Thread
- *2* RealtimeThread (but not an EventHandler)
- *2EDF*
- *3* NoHeapRealtimeThread (but not an event handler)
- *3EDF*
- *4* EventHandler that can be mapped as RealtimeThread
- *4EDF*
- *5* EventHandler that can be mapped as NoHeapRealtimeThread
- *5EDF*

Correct?

We think that it will be next to impossible to provide an implementation that allows all these possibilities! We think that it is impossible to demand a minimum priority-based implementation of RealtimeThread and at the same time allow EDF scheduling on RealtimeThread. We therefore propose restricting the use of EDF (and others) to some of the threads/handlers.

One proposal would be to allow EDF in all handlers but not in any thread. In other words, an EDF implementation of the standard would support:

- *1*
- *2*
- *3*
- *4EDF*
- *5EDF*

Note that this implies that synchronization between *2* and *4* is illegal. There is no practical way to synchronize between the EDF world and the priority world.

An alternative (less favored by us, but do-able) would be to allow EDF in the "NoHeap" world:

- *1*
- *2*
- *3EDF*
- *4*
- *5EDF*

This now has similar synchronization restrictions to what you already specify: *2* and *3* are illegal (and so are *4* and *5*). So maybe that is what you intended?

> **Bollella:** Confirmed. Synchronization is legal between any two of the five types plus java.lang.Threads. This is not to say that one should not do such synchronizations and expect deterministic behavior. So, our programming model strongly discourages such synchronization, but it is legal.
>
> The RTSJ does not comment on the possibility of multiple scheduling policies co-existing on and managing disjoint sets of threads. It is not proven that this is either possible or impossible. The RTSJ provides for such multiple scheduling policies to exist in a single implementation.

Question: In an EDF world, it is vital to have two distinct error handlers:

- Deadline miss
- Duration overrun

Your current proposal of "overrun" mostly covers "duration overrun" or "period overrun." We propose the addition of a "deadline miss" handler which is separate from the overrun handler.

Bollella: We have added deadlineMissHandlers where appropriate.

Question: [Reference to previous version removed] Here is an instance of potential confusion related to the point raised ... above. I don't know what "[a]ll priorities ... must be logically lower" means. If it is referring to the priority values ..., then it might mean they have higher execution eligibility. I suspect that the sentence was intended to mean "[a]ll instances of threads that are not instances of RealtimeThread, NoHeapRealtimeThread, or AsyncEventHandler classes or subclasses of those classes must have lower execution eligibility than any ready-to-execute thread instance of those classes." However, I am not convinced that this is a reasonable requirement. For one thing, again this is a priority-centric requirement. (What does it mean "require policies that do not use explicit priorities to determine execution eligibility"?) For another thing, even for priority-based policies, it seems unreasonable to imagine that the RT Java application has all of the most critical threads. What about threads that are part of the JVM (or the operating system) that should be able to run in preference to real-time Java thread instances? (I imagine that this requirement is supposed to capture the approach taken in some systems where there is a portion of the priority space denoted as "real-time priorities," where the real-time priorities are higher than all other priorities in the system -- typically including those of system threads. That's not a bad thing; it's just not the only thing.)

Bollella: Rewritten. java.lang.Thread priorities are from the same scale as javax.realtime.RealtimeThread priorities. We require 28 for RealtimeThreads and that these be above any the Threads can use. Note we do not restrict the assignment of any of the priorities to any of the threads.

Question: [Reference to previous version removed] The requirement -- "The dispatching mechanism must allow the preemption of the execution of schedulable objects at any time" -- is too strong. The final phrase is the problem. "At any time" is the right aspiration; however, it will not be realizable due to various sources of preemption latency and limitations on preemption granularity.

Bollella: Changed to, "The dispatching mechanism must allow the preemption of the execution of schedulable objects at a point not governed by the preempted object."

Question: [Reference to previous version removed] This requirement states that "[n]o part of the system may change the execution eligibility of a schedulable object for any reason other than implementation of a priority inversion algorithm." This is too restrictive. For example, a typical timesharing algorithm will lower the execution eligibility of compute-intensive threads. There is no reason to prohibit a real-time policy from performing a similar function. Also, what does "the system" refer to? The JVM? The JVM and the operating system? All of the software on a given processing node?

Bollella: This is our definition of fixed-priority. We preclude the system from changing the priorities (except as noted) because the programmer may have made the assignment according to some algorithm unknown to the system and based a feasibility assessment on such assignment. If the system changes the priority the feasibility assessment may become invalid. The system means everything except application logic.

Question: [Reference to previous version removed] This requirement states that "[a] blocked thread that becomes ready to run is added to the tail of any runnable queue for that execution eligibility." Again, this is a priority-centric requirement. It is perfectly reasonable to imagine scheduling policies that construct schedules when making scheduling decisions. When a thread becomes ready to run, there is no reason for a high-level requirement to specify where that policy must place that thread in terms of queues. The thread might be placed in the middle of a tentative schedule, corresponding to the middle (or even the front) of a specific queue; and the relationship between execution eligibility and runnable queues seems best left up to the scheduling policy. (Again there is a need for more explicit definitions: What is a "runnable queue?") Finally, is there a difference between a blocked thread and a preempted thread? (I assume so.) If so, this requirement would seem applicable to any thread that becomes ready to run; consequently, it should not mention only blocked threads.

Bollella: Fixed. We now note that these are for the required scheduler only.

Question: [Reference to previous version removed] This requirement states that "[a] thread whose priorityLevel is explicitly set by itself or another thread is added to the tail of the runnable queue for the new priorityLevel." Again, this is a priority-centric requirement. But even ignoring that point, there might be exceptions even for priority-based systems. For instance, if a thread that has had its execution eligibility increased by means of priority inheritance then explicitly changes its priority, satisfaction of this requirement could compromise the priority inheritance. (That might be perfectly acceptable behavior, but it could be argued that it is not the desired behavior. The point is that we might want to think about such cases when considering this particular requirement.)

Bollella: Fixed. We now note that these are for the required scheduler only.

Question: [Reference to previous version removed] This requirement states that "[a] thread that performs a yield() goes to the tail of its execution eligibility queue." Once again, placement of a thread in execution eligibility queues should be determined by the particular scheduling policy in effect. There is no single placement that is right for all reasonable scheduling policies. (Also, this and the previous two requirements use three different phrases to describe (I believe) the same thing: "runnable queue for that execution eligibility," "runnable queue for the new priorityLevel," and "execution eligibility queue." A single term should be adopted and used throughout.)

Bollella: Fixed as above and changed wording of 7,8,9 to be the same.

Question: [Reference to previous version removed] This requirement states "[the] scheduling policy and semantics given by 1-9 above and by the descriptions of the classes, methods, and their interactions are required in all implementations of this specification." I agree that there should be a set of high-level requirements that all implementations of this specification satisfy, but I think that the requirements in 1-9 are not it. It would be desirable to try to define a set of more general scheduling semantics and requirements. Then a second set of requirements could be specified for all priority-based policies. Or a set of requirements could be specified for a specific priority-based policy.

Bollella: We decided to leave it as it is.

Question: [Reference to previous version removed] The requirement closes with the statement that "[those] implementations that choose to not implement a feasibility algorithm may return success when any schedulable object requests admission." The word "may" seems too weak. Is it intended that it mean "can?" "Should?" "Must?"

Bollella: We are thinking about this one. The RI team is going to implement a feasibility test for a very restricted task model. I may make it a requirement as the RI work progresses.

Question: The rationale concludes with an argument for why only 28 priorities are used, rather than the 32 that some would argue is reasonable. It is claimed that 28 reflects a compromise that leaves priorities available for logic executing outside the JVM. It is good to recognize that those threads supported by the JVM might not be the only threads in the entire system. Four feels arbitrary as the number to allow for others. Is there a solid technical reason for that selection? If the JVM is hosted on an operating system that supports only 32 priorities, and the OS is assumed to be a given, I know of no assurance that the OS will only use four priorities. (POSIX feels that it can require a minimum of 32 priorities for real-time systems. Why should this specification differ from that number?)

Bollella: We are OK with it as is.

Question: POSIX standards are interesting for at least a few reasons: (a) they have been around for a number of years and have matured over that time, and (b) there are a number of real-time products that conform to the POSIX standards already. Consequently, I believe we should think carefully before differing from the POSIX specification in terms of priority-based scheduling behaviors. Since the POSIX specification is more mature than ours, we must be very certain that any behavior we want that differs from POSIX is justified. Each difference can have a price -- vendors might be in a position to have to support two different standard behaviors (one for POSIX and another for RT Java) and programmers might have to have two different scheduling models in their heads.

The RT Java specification never directly refers to the POSIX standards. Rather, it states the required behaviors explicitly. In principle, that is a reasonable thing to do; but the existing draft has several problems in that respect. These include:

1. The specification is incomplete. A few examples follow.

   Although it specifies the position in a queue of a newly unblocked thread and it talks about preemption, it never talks about the position in a queue of a preempted thread.

   Although it specifies that there are a minimum of 28 priority levels, there is no obvious way to determine how many there actually are. Moreover, there is no obvious way to determine what numeric values those priorities are (e.g., 1-28, 0-27, 50-77, ...).
2. The specification is impossible to satisfy. It might be that the wording is just a little bit too loose right now.

   For instance, Requirement 5 states that "[t]he dispatching mechanism must allow the preemption of the execution of schedulable objects at any time." While we might aspire to such a goal, it is almost certainly unrealizable. For example, the dispatcher cannot preempt a thread in the middle of its execution of a processor instruction. (I'm not trying to be difficult or silly here; I'm just trying to have the specification state an actual and satisfiable requirement.)
3. The specification differs from POSIX for no obvious reason.

   For instance, the minimum number of priorities must be 28 for RT Java and 32 for POSIX. The suggestion is that some priorities might be needed for the OS or the JVM and shouldn't be available for the RT Java programmer. I do not think this follows the usual strategy for real-time systems, where the programmer is given sharp tools -- tools that are sharp enough to cause system problems.

   A higher POSIX priority is reflected by a higher priority number. A higher RT Java priority is reflected by a lower priority number.
4. The specification includes requirements that are overly constraining for some scheduling policies.

   Requirement 6 states that "[n]o part of the system may change the execution eligibility of a schedulable object for any reason other than implementation of a priority inversion algorithm." (As an aside, I'm not completely certain about what constitutes the "system.") This requirement would seem to preclude the implementation of a standard timesharing policy, which will automatically reduce a CPU-bound thread's execution eligibility.

The current set of "Semantics and Requirements" attempt to describe behaviors in terms of execution eligibility, rather than strict priorities. Nonetheless, priority semantics are present in too many situations. For example, Requirement 7 specifies that "[a] blocked thread that becomes ready to run is added to the tail of any runnable queue for that execution eligibility." This is perfectly reasonable for "standard" priority-based policies that want to be fair within a single priority level. But it is presumptuous for us to dictate that an arbitrary scheduling policy must place a thread in a particular place in its run queue(s). Certainly a policy that would, in essence, populate a run queue with an ordered list of ready threads should be free to place a newly (re)runnable thread at any position in that list for whatever reason it sees fit.

So, what would I suggest about all of this?

Unless there is a compelling reason not to, I would like us to point to POSIX specifications for FIFO (and Round Robin?) scheduling behaviors. (I might even like us to have a set of methods that closely parallel the POSIX APIs for the relevant priority specifications and inquiries.) This would allow vendors implementing JVMs on POSIX-compliant platforms to take advantage of that fact and explicitly let them know that there were no incompatibilities in these two models. Moreover, I would adopt the minimum number of priorities used by POSIX (32) and would order priorities in the same way as POSIX.

I would try to break up the requirements into two groups. One group would talk about scheduling requirements that hold for all policies; the second group would specify specific scheduling policy behaviors for FIFO (and Round Robin?) priority policies. (I have the impression that the current list was trying to be the policy-independent set of requirements, but it has too many policy dependencies and assumptions in it.)

Bollella: In general we cannot say the same things as POSIX because we have other concerns (java.lang.Thread, native threads, Java processors, non-POSIX RTOSs, etc.) Thanks for your comments.

Question: I continue to imagine the situation in which a vendor wants to produce a package that will operate on RT Java platforms. The vendor wants the package to be widely applicable -- that is, it should run on multiple RTJ JVMs from different vendors. Since the current notion (of the Real-time for Java Experts Group) is that the JVM has a single scheduling policy and that policy is bound to the JVM, the vendor wants the package to be capable of running under multiple schedulers, potentially using several different scheduling policies. Consequently, the package is capable of supplying appropriate scheduling parameters for these policies -- say, priorities for one scheduling policy and deadlines for another.

What the package now needs is a means of determining the scheduling policy in effect on a given JVM. I do not see that capability in the specification (as I read it -- which is admittedly less than a complete understanding).

The getCurrentScheduler() method does not satisfy my need because it returns a reference to the scheduler, but does not indicate what scheduling policy that scheduler implements.

One potential approach is to use the createRealtimeParameters() method to get the right type of parameters and then fill them out as best you can. But, how do you do that? Is the RealtimeParameters data type somehow self describing? Even if it was, I might want to know a little bit more than just the field names.

In any case, regardless of how it is accomplished, there should be a high-level requirement that specifies that there be a way to determine the scheduling policy in effect on a JVM. (Looking to one possible future, it might best be phrased as an ability to learn the set of policies available and/or active on a given JVM. For the first version, there will only be a single policy.)

Bollella: We require the base scheduler, the APIs, PIP in synchronized. These give much more binary portability than the real-time systems industry has had prior to the RTSJ. We've cleaned the scheduling and parameter classes up a bit. The latest version is better.

Question: My comments in this area are motivated by a couple of factors. The most immediate is addressing the situation of a vendor that wants to write a single package that can be installed on any of a number of RT Java JVMs. However, a potentially more compelling motivation is the future growth path that I imagine for this technology.

If we have a scheduling framework that is tightly bound to a JVM, then our potential customers are tied to that JVM for their distributed real-time processing (because other JVMs might not provide the same scheduling policies). Some customers might be content with the ability to write a relatively small policy module to implement their optimal scheduling policy. They then use a JVM with that policy and spread it thoughout their enterprise.

At some point, I speculate that there will be some software that they cannot reasonably rewrite or modify to use their scheduling policy. This might be due to the fact that they do not have the source code available or the cost of acquiring it (or contracting another party for the work) is prohibitive. Or it might be difficult to manage that particular application with the scheduling policy that they had in mind. The software in question could be a communication package, a database package, or any other software that is essential, or maybe just desirable, for accomplishing their mission.

The JVM and its scheduling features could help if it could support multiple concurrent scheduling policies. I really imagine that this will be a common case in the future. My experience with the MK7.3 operating system at The Open Group Research Institute contributes to that feeling. There, the microkernel was altered to support a scheduling framework and a variety of scheduling policies. But since there was a great deal of legacy code in the system (e.g., OSF/1 Unix personality, X Windows System, thread libraries, applications, and so on), the initial design had to include the ability to support multiple concurrent scheduling policies. This was necessary for the "average" real-time user because the microkernel and OS server were written with certain scheduling assumptions in mind. If the user wanted to use any other policy, it would have to be used in addition to those that were already present. (Few projects will be able to abandon all of the legacy code they have in order to start from scratch.)

Also, whether we explicitly address this point or not, we are typically already in a situation where we have multiple concurrent scheduling policies on our JVM-based platforms. That is, as I understand things, the typical case is that the RT Java JVM is hosted on an OS, which, of course, has its own threads and its own scheduling policy(ies). Regardless of what goes on inside the JVM, in the end the OS-supplied scheduling facilities will schedule all of the threads on the machine -- and that could well include all of the deadline-scheduled threads in a JVM and all of the priority-based threads in the OS and other concurrent applications (if any). The Real-time for Java Experts Group is essentially saying that this coexistence is necessary. Moreover, it is saying that the JVM can solve it adequately, using the underlying OS facilities. Yet it is not willing to allow the JVM to support exactly that same sort of scheduling policy mix -- limiting the ability of customers to utilize legacy software and application-specific scheduling policies.

> **Bollella:** The latest version should address all of the issues raised here.

Question: Would it be clearer if we renamed ProcessingGroupParameters to SchedulingGroupParameters ?

> **Bollella:** I'm open to changing the name of ProcessingGroupParameters but I don't want to imply that these parameters have anything to do with scheduling. They are release characteristics. How about GroupReleaseParameters?

Question: Currently AsyncEvent has no mechanism to deal with data associated with an event. While this works well for most POSIX events, many embedded system events have associated data that must be evaluated to determine the appropriate action. This evaluation could certainly be done by the AsyncEventHandlers, but it raises the question of how to get the data to the AsyncEventHandler.

For instance, controlling one of the serial ports on a PowerPC processor is a relatively simple and common embedded task. But in order to do this the BufferDescriptor must be checked for several flags each time an interrupt is received from the port. The BufferDescriptor register provides status and error condition information about the port each time an interrupt occurs. The actions of an AsyncEventHandler in this case would vary based upon the contents of that register.

> **Bollella:** Thanks for your comment. We've talked about this a lot. If we add any data at all to AE and AEH we will have to make the underlying mechanism much more heavyweight (queues, sequence numbers, priority will affect execution order, etc.). We prefer to leave it as it is and let folks that need to pass data build such an event mechanism on top of AE and AEH. In this way we support both requirements (i.e., systems with a few heavyweight events and systems with thousands of lightweight events).

Question: The spec says under "Thread Scheduling and Dispatching" that "The specification is constructed to allow implementations to provide unanticipated scheduling algorithms." Section 3.1, point 6 seems to inhibit this.

> **Bollella:** We fixed point 6. For schedulable objects managed by the required scheduler no part of the system may change the execution eligibility for any reason other than implementation of a priority inversion algorithm. This does not preclude additional schedulers from changing the execution eligibility of schedulable objects -- which they manage -- according to the scheduling algorithm.

Question: Isn't it possible to remove the redundancies between PeriodicParameters and ProcessingParameters? Especially for RealtimeParameters whose relParams are PeriodicParameters. Seems to only make the system harder to use with no perceived advantage.

> **Bollella:** Thanks for your insight. We have modified the hierarchy of the Parameters objects and the redundancies are gone.

Question: How are the second bulleted item ("Manage the assignment of execution eligibility to schedulable objects.") and the seventh bulleted item ("Assign execution eligibility values to schedulable objects.") related? They sound like they might be referring to the same thing. Or one might refer to the association of values with threads, while the other describes the evaluation of those values to determine execution eligibility. Or they might have some other relationship. In any case, the meaning is not clear. I would suggest combining them into a single item or sharpening them up to make a clear distinction.

> **Bollella:** Assign implies that application code uses setPriority() to assign the priority based on application logic. Manage implies that implementations have the system assign priorities as in RMA.

Question: It appears the ASE handlers can also be periodic, sporadic or aperiodic (this is defined by the RealtimeParameters passed on the handler creation).

ASE handlers are schedulable objects. In Section 3 it says that the Scheduler performs feasibility analysis and admission control over a set of SCHEDULABLE objects. However, there is no AdmissionControlException associated with ASE or their handlers.

> **Bollella:** You are right. We've added the necessary exceptions and text.

Question: Does the fifth bulleted item ("Manage the execution of instances of the AsyncEventHandler and RealtimeThread classes") imply that scheduling instances of other thread classes is not the responsibility of the scheduling facility?

> **Bollella:** Correct.

Question: The third paragraph states that "[a] schedulable object is considered to have the execution eligibility of the instance of the SchedulingParameters class given to the constructor of the schedulable object." The sentence suggests to me that the SchedulingParameters class is intended to capture all of the information necessary to determine execution eligibility. I believe that this might be possible for some scheduling policies (e.g., priority-based scheduling), but will not be true in general. For instance, a scheduling policy that uses accrued processor time in making a scheduling decision (such as minimum laxity (slack) scheduler or a timesharing policy that penalizes CPU-intensive threads) will be using that information that is (I assume) not part of the SchedulingParameter class. Even for priority-based policies, priority alone does not totally capture what I intuitively assume execution eligibility means. For example, in a uniprocessor system where threads of a given priority are placed in a FIFO queue, the execution eligibility of a given thread is a function of its priority (which is part of the SchedulingParameters class) and its position in a run queue (which is presumably not part of the SchedulingParameters class). In addition, if priority inheritance is used by the scheduling policy, the instantaneous priority of a thread can be higher than its "assigned" priority. This might or might not be captured in the SchedulingParameters class.

> **Bollella:** Such implementations will subclass Scheduler and Parameters objects as appropriate.

Question: The preceding question points out the need for a more careful description of several key terms. In particular, I didn't see definitions for execution eligibility or for periodic, aperiodic, or sporadic thread executions (or scheduling parameters).

> **Bollella:** We've added some explanation.

Question: The final sentence of the introductory section ("The demand may be determined, in the case of PeriodicParameters or SporadicParameters, statically, or in the case of AperiodicParameters, dynamically.") refers to static and dynamic determinations of demand. The terms static and dynamic are not defined though. I don't believe that statically is intended to mean offline, although it might. Again, definitions of periodic, aperiodic, and sporadic would probably help.

> **Bollella:** Fixed in latest version.

Question: On enumerated point (1): POSIX requires 32 unique priorities for real-time scheduling. Why deviate from POSIX specification by requiring only 28? (I also raise this point with respect to the "Rationale" section.) Also, I see no requirement that a program be able to determine how many priorities are actually supported by a given Scheduler or what their (absolute) values are. Finally, this is a very priority-centric requirement. That is, for instance, what meaning would this requirement hold for an Earliest Deadline First scheduler?

> **Bollella:** Explained in rationale. RealtimeThread.MAX_PRIORITY - RealtimeThread.MIN_PRIORITY + 1 = number of prior levels. All implementations are required to have the base scheduling mechanism (28 unique priority levels, fixed-priority-preemptive). Implementations are free to provide other scheduling mechanisms (such as EDF). They do this by subclassing Scheduler and Parameters classes, adding necessary fields and setting RealtimeThread.scheduler = subclassedScheduler.instance(). for those realtime threads that will be scheduled by the additional scheduler.

Question: On enumerated point (2): I believe that this convention is the opposite of that adopted by POSIX, where higher priorities are denoted by higher priority numbers. (My concern is not that one way is right and the other is wrong. After all, it is just a convention. However, the POSIX standard has been around for several years and is known. This standard is just being drafted and examined. Deviations from well known standards should be well motivated. By the way, I have worked extensively on a system that uses the lower-priority-number-means-higher-priority convention, and it definitely complicates conversations concerning scheduling parameters and behaviors.)

> **Bollella:** Fixed. There were other folks who raised this point. (I personally disagree but agreed to go with the majority of the Experts Group.)

Memory Management

Question: We have strong concerns about your GarbageCollector class: we think that no matter how you define the API, there will always be numbers that are completely irrelevant or impossible to specify for a given GC algorithm. Maybe allow the return of a value that means "for the algorithm used, this number is meaningless" ?

> **Bollella:** We've added a NOT_APPLICABLE field which implementations are free to return for any of the get<>() methods in GC.

Question: "The finalizers for each object in the memory associated the instance of ScopedMemory are executed to completion before any statement in any thread is executed." This means an implicit synchronization of the actual thread to all other threads, which seems to be unacceptable.

> **Bollella:** Made this relative to access to the memory area. (That is, finalizers must complete before any statement attempts to access the memory area.)

Question: MemoryArea.newArray(java.lang.Class T, int n) It is not specified whether T is the array type or the element type. The restrictions that T may not be interface, abstract, or array is wrong. If T is the array type it must be an array; if T is the element type it can be any type.

> **Bollella:** Fixed.

Question: The name of this memory pool is misleading. Any memory allocated for use in Java must be initialized to zero, which is an operation that requires time linear in the size of the allocated memory.

A better name would be LTMemory, standing for linear-time allocation. This is the best a Java implementation might guarantee.

LTMemory will also allow more sophisticated automatic memory management tools to be applied. A garbage collector that guarantees linear time overhead on allocation might even use memory from the normal collected pool as LTMemory.

> **Bollella:** Based on your comment we are revisiting the semantics of CTMemory. Check the Web site periodically for updates.

Question: "Ready to execute" should be defined somewhere. In addition, blocked and preempted should also be defined.

> **Bollella:** Preemption maybe. Remember everything the RTSJ says tends to drive it toward being efficiently implementable on only certain platforms, so we want to walk a fine line between saying too much and too little.

Question: You have implemented getter and setter methods for the primitive integer data types. Why have you not implemented getter and setter methods for the floating-point types as well? Real-time process control systems implemented in Java will require access to floating-point variables. If floating-point primitive types are not supported, how do you propose that Java applications gain access to these floating-point variables?

> **Bollella:** We have decided to specify a subclass of RawMemoryAccess (tentatively named RawMemoryFloatAccess) to implement get/set for float and double data types on physical memory. The class is optionally required. The spec says it must be implemented if and only if the underlying Java platform supports floating-point data types.

Synchronization

Question: You already specify that certain synchronizations are illegal. But currently it is not clear what happens if the user writes an illegal synchronization in the code. We propose that an exception should be thrown for such illegal synchronizations.

> **Bollella:** Synchronizations between any thread types are legal, although maybe not a good idea.

Question: The synchronization discussion, both here and Chapter 1, seems to have overlooked the behavior of wait/notify with regard to queuing. Presumably threads will queue in priority order with FIFO within priority as for acquiring monitors. A notify() then releases the highest priority thread.

> **Bollella:** The list in section 1.3.3 about queuing order is meant to include wait/notify. It has been rewritten to generalize the requirements away from priority-based systems.

Time

Question: I assume from the spec that there can be multiple instantiations of AbsoluteTime. My team and I have had long discussions about this and the assumption is that there can be multiple instantiations of AbsoluteTime, but shouldn't there also be a Singelton RealtimeClock. A RealtimeClock Singleton would provide the interface to set and get the system clock for an embedded device. So the question is, did the spec not wish to go as far as defining a RealtimeClock, or was AbsoluteTime expected to provide that functionality? If AbsoluteTime is expected to provide that interface, should it be a Singleton?

> **Bollella:** You are right. The RTSJ assumes there is a default Clock. (Clock.getRealtimeClock()). The RTSJ also

allows the possibility of other Clocks.

Timers

Question: Should setResolution in Clock take a parameter of RelativeTime rather than HiResTime?

> **Bollella:** Relative time.

Question: Where are the subclasses of Clock defined? At the moment we only seem to have an abstract class. Surely, we should have at least one non-abstract class defined?

> **Bollella:** The spec only provides for a singleton default Clock (Clock.getRealtimeClock()). Implementers and application developers should subclass the Clock class.

Question: What Clock is the sleep related to? I don't understand how the Thread class is tied to a particular clock.

> **Bollella:** RealtimeThread.sleep() is decremented by the default Clock (Clock.getRealtimeClock()). We added RealtimeThread.sleep(Clock,HighResolutionTime) to generalize the notion to subclasses of Clock.

Asynchrony

Question: The action in point 7 on page 16, regarding wait(), sleep() and join() is completely wrong in my view. It states that if interrupt occurs in a non-AI method that is doing a sleep(), wait() or join(), then the interrupt bit gets set and control continues normally until an AI method is entered. This changes the semantics of non-real time interrupts and would break code that real-time threads could be trying to use. Well written non-RT code will deal with interrupts by throwing the InterruptedException and/or returning prematurely when the interrupt is detected(i.e.. interruption == cancellation). If sleep(), wait() or join() throw IE then, by convention, the interrupt bit is cleared. In my view, under RTJ, interrupt should have a composite effect: it should trigger a normal interruption and an async-interruption. If the normal interruption is dealt with first then the async interruption will be dealt with as soon as control enters an AI method. If the AIE is dealt with first then it consumes the normal interrupt. In other words sleep(), join() and wait() should continue to operate exactly as they are currently defined to do:

- If a non-AI method invokes them and interrupt is called, then an InterruptedException is thrown. When control enters an AI method the pending AIE is thrown (replacing the IE if necessary).
- If an AI method invokes them and interrupt is called, then an AIE is thrown and dealt with accordingly

These semantics make much more sense to me because they ensure the continued responsiveness of well-written non-RT code, that may be used by RT-code.

> **Bollella:** Your comment on point 7 is correct and we've changed the text to read as you suggest.

Question: "For critical handlers that can" -> "For critical handlers that can NOT" I think?

> **Bollella:** Yes, you are correct. Changed.

Question: The relationship between the run method and the handleAsyncEvent method is not clear. If run calls handleAsyncEvent then this should be stated. Perhaps it should be made final?

> **Bollella:** run() in AEH is now final. We cleaned up the text.

Question: The foils given out at Real-Time Systems Symposium by Greg say that if deadline is set to 0, this means deadline = period. I hope this is not still the case. It ought to be possible to generate an asynchronous event if a deadline overruns.

> **Bollella:** Text now indicates that deadline is from the start of the period. Default = period. All appropriate classes now include an AsyncEventHandler deadlineMissHandler field.

Question: It is not clear what happens if doInterruptible is called by more than one thread and then a call to fire is made.

What is the relationship between AIEs and ThreadGroups? Ideally, I would like to be able to send an AIE to a ThreadGroup (as I can interrupt a thread group).

The rules for multiple AIE on the same thread on the same thread are not clear:

Consider the following example:

```
public class NestedATC
{
  AsyncnhronouslyInterruptedException aie1 = new
     AsyncnhronouslyInterruptedException();
  AsyncnhronouslyInterruptedException aie2 = new
     AsyncnhronouslyInterruptedException();
  AsyncnhronouslyInterruptedException aie3 = new
     AsyncnhronouslyInterruptedException();

  public void method1() throws AsyncnhronouslyInterruptedException
  {

    // ATC-deferred region

    // ATC allowed
  }

  public void method2() throws AsyncnhronouslyInterruptedException
  {
    method1();
  }

  public void method2() throws AsyncnhronouslyInterruptedException
  {
    method2();
  }
```

Now suppose that a thread t has created an instance of NestedATC and is active in the ATC-deferred region of method 1. Now suppose while it is there, it is interrupted by all three AIEs. What happens?

> **Bollella:** Text and APIs now restrict only one invocation of doInterruptible to be active at one time (this was always our intent). So cross thread or nested calls to doInterruptible are not allowed. Also, enable() and disable() are valid *only* from within a doInterruptible. fire() returns false if there is no current doInterruptible or if there is a current doInterruptible and disable() has been called.

Question: Should the wait method in Object be overridden somewhere so that it takes a parameter of HiResTime?

> **Bollella:** When we discussed this issue we concluded that one should use RealtimeThread.sleep(highResolutionTime). However, as I write this I realize that is not what you were asking for. We should discuss it again and I'll get back to you on this one.

Question: Should HiResTime say "implements comparable" somewhere?

> **Bollella:** Done. We had this in an earlier version but took it out because Comparable is not in Java 1.1.x. We decided to put it back and have implementers add one if they implement on a Java 1.1.x base.

Question: Should it be possible to handle a deadline overrun?

> **Bollella:** We added an AsyncEventHandler deadlineMissHandler field to the appropriate classes.

Question: The specification indicates that when interrupt occurs during a blocking java.io operation that is within an AI method, then the AIE should be thrown. This statement seems to imply that the existing interrupt semantics of IO are simply being extended to throw AIE. However the existing notion of interruptible IO has actually been deprecated by Sun ... Apart from implementation problems on some platforms there is the whole issues of semantics: what does interruptible IO mean? what state is the IO stream left in after an interrupt? How can an application perform guaranteed IO? These are non-trivial issues that seem to have been overlooked in the preparation of the spec. Even if the spec team comes up with answers to these questions, each RT-JVM will have to completely re-implement the java.io package to make it work.

> **Bollella:** By interruptible I/O we mean simply that I/O calls that block waiting for the completion of an I/O operation may complete abnormally (i.e., control resumes at the appropriate catch clause. The I/O stream is left as though the call never occurred (i.e., the completion of the call is atomic as viewed from the application). I don't know what guaranteed I/O is. We don't agree that java.io will have to be completely re-implemented. But whatever the costs, the real-time community needs this function.

Question: I thought I understood the general AIE issue and how things would work, but the AIE API has left me thoroughly confused. The documentation for this class is very poor -- there is no information on how it is supposed to be used. I would have expected a method like doInterruptible() to be static but it is an instance method -- why? What is that supposed to mean? What does it mean to be able to enable and disable a specific AIE? Exactly what is the notion of a "current AIE" (there is some mention of nested AIEs in Chapter 1 but it is unclear how that discussion pertains to the actual API)? How does propagation work? Why do you "fire" an AIE -- exceptions are thrown not fired.

There seem to be some complex nesting semantics and per-interrupt control that is simply not documented.

> **Bollella:** We have modified the AIE API a bit since 0.9 and have added more explanation.

Question: This method does not pass any data to the asynchronous event handler; it just makes it ready to run. I have previous experience with a facility like this where it was advantageous to pass information to the handler. This experience may or may not be relevant to this case, but consider an example. Imagine a scheduling policy that receives time constraints from the application for each Realtime Thread. There is an associated asynchronous event handler that is executed in case a time constraint is not satisfied. If nested time constraints are possible (and they would certainly be desirable in an object-oriented real-time system where there are a variety of time granularities of application and device timing requirements), then a number of different approaches are possible. For instance, an input parameter to the handler could notify it of which time constraint was unsatisfied. Or the parameter could indicate the time constraint under which the handler should execute. Or the parameter could refer to a critical data structure that can be consulted to extract a partial result or provide other information to the application.

> **Bollella:** We will keep our current dataless model. We do have an explanation in the specification.

System and Options

Question: Requirement 2: To be compatible with POSIX and Ada, and avoid confusion, it would be nice if high numbers meant higher execution eligibility.

> **Bollella:** We had other comments about this. It has been changed.

Exceptions

Question: MemoryAccessError "The exception thrown when an attempt to access out-of-scope memory is detected." Such a situation can never happen when the reference assignment rules (4.1.15) are enforced by the system. If this is the exception thrown in case of a violation of the reference assignment rules it should be named and documented accordingly.

> **Bollella:** We are not closed on this point yet.

Resources

- See the current Real-time Specification for Java (RTSJ).

About the author

Greg Bollella received a Ph.D. in Computer Science from The University of North Carolina at Chapel Hill in 1997. His dissertation was based on an architecture he created to add support and analysis for real-time computation -- including earliest-deadline first scheduling and rate-monotonic prioritiy assignment -- to existing, general-purpose operating systems. Greg joined IBM in 1991. His most recent assignments included leading the team that created the real-time Java robot demonstration displayed at the Embedded Systems Conference in San Jose in November 1998 and leading the Real-Time Java Expert Group under the Java Community Process (JSR-000001).