The Real-Time Specification for Java

The RTSJ provides a platform that will let programmers correctly reason about the temporal behavior of executing software. Two members of the Real-Time for Java Experts Group explain the RTSJ's features and the thinking behind the specification's design.

Greg Bollella
IBM

James Gosling
Sun Microsystems

New languages, programming disciplines, operating systems, and software engineering techniques sometimes hold considerable potential for real-time software developers. A promising area of interest—but one fairly new to the real-time community—is object-oriented programming. Java, for example, draws heavily from object orientation and is highly suitable for extension to real-time and embedded systems.

Recognizing this fit between Java and real-time software development, the Real-Time for Java Experts Group (RTJEG) began developing the real-time specification for Java (RTSJ)[1] in March 1999 under the Java Community Process.[2] The goal of the RTJEG, of which we are both members, was to provide a platform—a Java execution environment and application program interface (API)—that lets programmers correctly reason about the temporal behavior of executing software. Programmers who write real-time systems applications must be able to determine a priori when certain logic will execute and that it will complete its execution before some deadline. This predictability is important in many real-time system applications, such as aircraft control systems, military command and control systems, industrial automation systems, transportation, and telephone switches.

We began our effort by looking at the Java language specification[3] and the Java virtual machine specification.[4] For each feature of the language or runtime code, we asked if the stated semantics would let a programmer determine with reasonable effort and before execution the temporal behavior of the feature (or of the things the feature controls) during execution. We identified three features—scheduling, memory management, and synchronization—that did not allow such determination. Requirements defined in a workshop sponsored by the National Institute of Science and Technology (NIST) combined with input from other industry organizations helped us identify additional features and semantics.

We decided to include four of these additional features: asynchronous event handling, asynchronous control transfer, asynchronous thread termination, and access to physical memory. These plus the three features from our review of the Java and JVM specifications gave us seven main areas for the RTSJ. We believe these seven areas provide a real-time software development platform suitable for a wide range of applications. The "How Well Does the RTSJ Meet Its Goals?" sidebar gives more background on the requirements and principles that drove RTSJ development.

The "RTSJ Timetable" sidebar lists important milestones and contacts for those interested in reviewing the draft specification. The RTSJ completed public review in February 2000, but the Java Community Process mandates that we not finalize the specification until we complete the reference implementation and test suites. As the community gains experience with the RTSJ, small changes to the specification may occur. Our goal in writing this article is to provide insights into RTSJ's design rationale—particularly how the NIST requirements and other goals affected it—and to show how this important specification fits into the overall strategic direction for defining an object-oriented programming system optimized for real-time systems development.

SCHEDULING

The Java specification provides only broad guidance for scheduling:

> When there is competition for processing resources, threads with higher priority are generally executed in preference to threads with lower priority. Such preference is not, however, a guarantee that the highest priority thread will always be running, and thread priorities cannot be used to reliably implement mutual exclusion.

Obviously, if you are writing software with temporal constraints, you need a stronger semantic statement about the order in which threads should execute. The typical way to define these semantics is through algorithms that determine how to choose the next thread for execution. The RTSJ specifies a minimum scheduling algorithm, which must be in all RTSJ implementations.

Minimum requirements

At the very least, all implementations must provide a fixed-priority preemptive dispatcher with no fewer than 28 unique priorities. "Fixed-priority" means that the *system* does not change thread priority (for example, by aging). There is one exception: The system can change thread priorities as part of executing the pri-

How Well Does the RTSJ Meet Its Goals?

In creating and refining the RTSJ, the Real-Time for Java Experts Group (RTJEG) used a set of core requirements based on a 1998-1999 workshop sponsored by the National Institute of Technology (NIST). We also used a set of guiding principles we established on the basis of feedback from the real-time community.

NIST core requirements

The NIST workshop's aim was to develop requirements for supporting real-time programming on the Java platform. The final workshop report, published in September 1999 (L. Carnahan and M. Ruark, eds., "Requirements for Real-Time Extensions for the Java Platform," http://www.nist.gov/rt-java), defines nine core requirements:

1. The specification must include a framework for the lookup and discovery of available profiles.
2. Any garbage collection that is provided shall have a bounded preemption latency.
3. The specification must define the relationships among real-time Java threads at the same level of detail as is currently available in existing standards documents.
4. The specification must include APIs to allow communication and synchronization between Java and non-Java tasks.
5. The specification must include handling of both internal and external asynchronous events.
6. The specification must include some form of asynchronous thread termination.
7. The core must provide mechanisms for enforcing mutual exclusion without blocking.
8. The specification must provide a mechanism to allow code to query whether it is running under a real-time Java thread or a non-real-time Java thread.
9. The specification must define the relationships that exist between real-time Java and non-real-time Java threads.

As Table A shows, the RTSJ satisfies all but the first requirement, which is not relevant because the RTSJ does not include the notion of profiles. Access to physical memory is not part of the NIST requirements, but industry input led us to include this feature.

Table A. How RTSJ features satisfy the NIST core requirements.

	NIST core requirements								
RTSJ features	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**
Scheduling	N/A		S					S	
Memory management	N/A	S	S						
Synchronization	N/A		S				S		S
Asynchronous event handling	N/A			S	S				
Asynchronous transfer of control	N/A								
Asynchronous thread termination	N/A		S			S			
Physical memory access	N/A								

Guiding principles

The guiding principles helped the RTJEG decide on what features and semantics to include as well as how to choose among alternatives. Below are the original guiding principles and an explanation of how the RTSJ satisfies them. The main text describes the RTSJ features and semantics in more detail.

Applicability to particular Java environments: The RTSJ must not include specifications that restrict its use to particular Java environments, such as a particular version of the JDK, the Embedded Java Application Environment, or the Java 2 Micro Edition.

Backward compatibility: The RTSJ must not prevent existing, properly written, non-real-time Java programs from executing on RTSJ implementations.

The RTSJ departs somewhat from the

ority inversion avoidance algorithm (which preserves priority inheritance). Without exception, *threads* can change their own or another thread's priorities.

The application program must see the minimum 28 priorities as unique; for example, it must know that a thread with a lower priority will never execute if a thread with a higher priority is ready. Thus, you cannot simplistically map 28 priorities into, say, a smaller number because of the underlying system. You can, however, implement the RTSJ on a platform that provides fewer than 28 native priorities. It would then be up to you to use whatever means available to provide the appearance of uniqueness.

Why *28* priorities? We chose this number because real-time scheduling theory indicates that you can expect close to optimal schedulability with 32 priorities.[5] We chose to leave four priorities for other tasks because the RTSJ will likely be only a part of the system, and some systems have only 32 native priorities.

You can, of course, add scheduling algorithms to this minimum requirement. The RTSJ provides three classes—Scheduler, SchedulingParameters, and ReleaseParameters—and subclasses of these that encapsulate temporal requirements. These classes are bound to schedulable objects (threads or event handlers). The required scheduler, an instance of the class PriorityScheduler, uses the values set in these parameter objects.

first principle in one aspect. We provide two Thread subclasses because localizing the notions of scheduling, events, and asynchronous transfer of control in the two real-time thread classes provides a more understandable and consistent specification. Java 2 Micro Edition profiles may exclude threads, and thus such profiles would not support RTSJ implementations. However, we feel excluding threads so changes the characteristics of an implementation that other RTSJ features would not be useful by themselves. If such profiles become widely available, we may consider subsets of the RTSJ.

Write once, run anywhere: The RTSJ should recognize the importance of WORA, but must also recognize the difficulty of achieving it for real-time programs.

This principle caused us to strengthen existing semantics rather than invent new ones. In so doing, we have allowed a much closer integration within the JVM and between the standard Java mechanisms and the mechanisms the RTSJ requires. Of course, we could have simply required essentially two JVMs per implementation to satisfy this principle, but we felt the tighter integration would produce cleaner, more robust, more flexible implementations. This principle also influenced our decision to add to the semantics of the synchronized keyword rather than adding a new semaphore class.

Observing this principle early gave us the freedom to write the specification so that implementers can choose among various scheduling and garbage-collection algorithms. We feel that this flexibility makes the RTSJ applicable to a wider range of real-time systems.

Current practice versus advanced features: The RTSJ should address current real-time system practice as well as allow implementers to easily add more advanced features in the future.

The requirement to use priority scheduling, as well as the use of no-heap real-time threads, and linear-time memory lets programmers closely adhere to the currently accepted real-time system development model. The flexibility of allowing implementers to include alternative scheduling and garbage-collection algorithms supports more advanced application programming models.

Predictable execution: The RTSJ shall hold predictable execution as first priority in all trade-offs; this may sometimes be at the expense of typical general-purpose computing performance measures.

Requiring asynchronous event handlers to have the scheduling semantics of threads shows that predictability is our first priority. We can envision very efficient implementations of the asynchronous event handling mechanism, but most likely all will have more overhead than simply executing the handler in the context of the current thread. We feel this is necessary because without it the execution of handlers could consume significant processor time and be invisible to the scheduler.

No syntactic extension: To make the tool developer's job easier (and thus increase the likelihood of timely implementations), the RTSJ must not introduce new keywords or make other syntactic extensions to the Java language.

Almost every feature of the RTSJ came under this principle. It is always easier to add new keywords, but we worked long and hard on many features to implement them with existing syntax.

Variation in implementation trade-offs: RTSJ implementations can vary in many decision-efficient or inefficient algorithms, actual clock resolution, inclusion of scheduling algorithms not in the minimum requirements, and variation in code path length for the execution of byte codes. The RTSJ should not mandate algorithms or values for these implementations but, rather, require that implementers meet the specification's semantic requirements. The RTSJ offers implementers the flexibility to create implementations suited to their customers' requirements.

Although it seems natural for a real-time specification to require some level of temporal determinism, we avoided it except for one case. We believe that mandating even the most seemingly reasonable determinism requirement narrows the specification's applicability because it may preclude less expensive implementations. We do, however, understand that real-time programmers need to understand the deterministic limits of the platform for which they design systems. Thus, the RJSJ imposes a documentation requirement for certain features. Implementers will have to provide documentation stating values for defined metrics that give the programmer enough information to construct correct systems.

RTSJ Timetable

December 1998: Sun Microsystems announces the Java Community Process and asks IBM to lead the real-time effort.

March 1999: Development begins.

September 1999: Participant review (all participants are Java licensees and members of the Java Community Process).

December 1999: Draft RTSJ released for public review.

February 2000: Public review closes.

June 2000: The RTSJ becomes available in book form (G. Bollella et al., *The Real-Time Specification for Java*, Addison Wesley Longman, Reading, Mass., June 2000.)

Late 2000: The finalized RTSJ, reference implementation, and test suites expected to be available.

The RTSJ is publicly available at http://www.rtj.org. The site also has news and updates. Comments and questions are welcome. Send them to comments@rtj.org.

Thread creation

The RTSJ defines the RealtimeThread (RT) class to create threads, which the resident scheduler executes. RTs can access objects on the heap and therefore can incur delays because of garbage collection. Another option in creating threads is to use a subclass of RT, NoHeapRealtimeThread. NHRTs cannot access any objects on the heap, which means that they can run while the garbage collector is running (and thus avoid delays from garbage collection). NHRTs are suitable for code with a very low tolerance of nonscheduled delays. RTs, on the other hand, are more suitable for code with a higher tolerance for longer delays. Regular Java threads will do for code with no temporal constraints.

MEMORY MANAGEMENT

Garbage-collected memory heaps have always been considered an obstacle to real-time programming because the garbage collector introduces unpredictable latencies. We wanted the RTSJ to require the use of a real-time garbage collector, but the technology is not sufficiently advanced. Instead, the RTSJ extends the memory model to support memory management in a way that does not interfere with the real-time code's ability to provide deterministic behavior. These extensions let you allocate both short- and long-lived objects outside the garbage-collected heap.

There is also sufficient flexibility to use familiar solutions, such as preallocated object pools.

Memory areas

The RTSJ introduces the notion of a *memory area*—a region of memory outside the garbage-collected heap that you can use to allocate objects. Memory areas are not garbage-collected in the usual sense. Strict rules on assignments to or from memory areas keep you from creating dangling pointers, and thus maintain Java's pointer safety. Objects allocated in memory areas may contain references to objects in the heap. Thus, the garbage collector must be able to scan memory outside the heap for references to objects within the heap to preserve the garbage-collected heap's integrity. This scanning is not the same as building a reachability graph for the heap. The collector merely adds any reference to heap objects to its set of pointers. Because NHRTs can preempt the collector, they cannot access or modify any pointer into the heap.

The RTSJ uses the abstract class MemoryArea to represent memory areas. This class has three subclasses: physical memory, immortal memory, and scoped memory. Physical memory lets you create objects within memory areas that have particular important characteristics, such as memory attached to a nonvolatile RAM. Immortal memory is a special case. Figure 1 shows how object allocation using immortal memory compares to manual allocation and automatic allocation using a Java heap. Traditional programming languages use manual allocation in which the application logic determines the object's life—a process that tends to be time-consuming and error-prone. The one immortal memory pool and all objects allocated from it live until the program terminates. Immortal object allocation is common practice in today's hard real-time systems. With scoped memory, there is no need for traditional garbage collection and the concomitant delays. The RTSJ implements scoped memory through either the MemoryParameters field or the ScopedMemory.enter() method.

Scoped memory

Figure 1c shows how scoped memory compares to immortal memory and other allocation disciplines. Scoped memory, implemented in the abstract class ScopedMemory, lets you allocate and manage objects using a memory area, or *syntactic scope*, which bounds the lifetime of any objects allocated within it. When the system enters a syntactic scope, every use of "new" causes the system to allocate memory from the active memory area. When a scope terminates or the system leaves it, the system normally drops the memory's reference count to zero, destroys any objects allocated within, and calls their finalizers.

You can also nest scopes. When the system enters a nested scope, it takes all subsequent allocations from the memory associated with the new scope. When it exits the nested scope, the system restores the previous scope and again takes all subsequent allocations from that scope.

Two concrete subclasses are available for instantiation as MemoryAreas: LTMemory (LT for linear time) and VTMemory (VT for variable time). In this context, “time” refers to the cost to allocate a new object. LTMemory requires that allocations have a time cost linear to object size (ignoring performance variations from hardware caches or similar optimizations). You specify the size of an LTMemory area when you create it, and the size remains fixed.

Although VTMemory has no such time restrictions, you may want to impose some restrictions anyway to minimize the variability in allocation cost. You build a VTMemory area with an initial size and specify a maximum size to which it can grow. You can also opt to perform real-time garbage collection in a VTMemory area, although the RTSJ does not require that.

If you decide to do this, you can build the VTMemory object with a garbage collection object to specify an implementation-specific garbage-collection mechanism. However, if you implement VTMemory, NHRTs must be able to use it.

Because, as Figure 1c shows, control flow governs the life of objects allocated in scoped memory areas, you must limit references to those objects. The RTSJ uses a restricted set of assignment rules that keep longer-lived objects from referencing objects in scoped memory, which are possibly shorter lived. The virtual machine must detect illegal assignment attempts and throw an appropriate exception when they occur.

The RTSJ implements scoped memory in two separate mechanisms, which interact consistently: the ScopedMemory.enter() method and the memory area field of MemoryParameters. You specify either one when you create the threads. Thus, several related threads can share a memory area, and the area will remain active until the last thread has exited. This flexibility means that the application can allocate new objects from a memory area that has characteristics appropriate either to the entire application or to particular code regions.

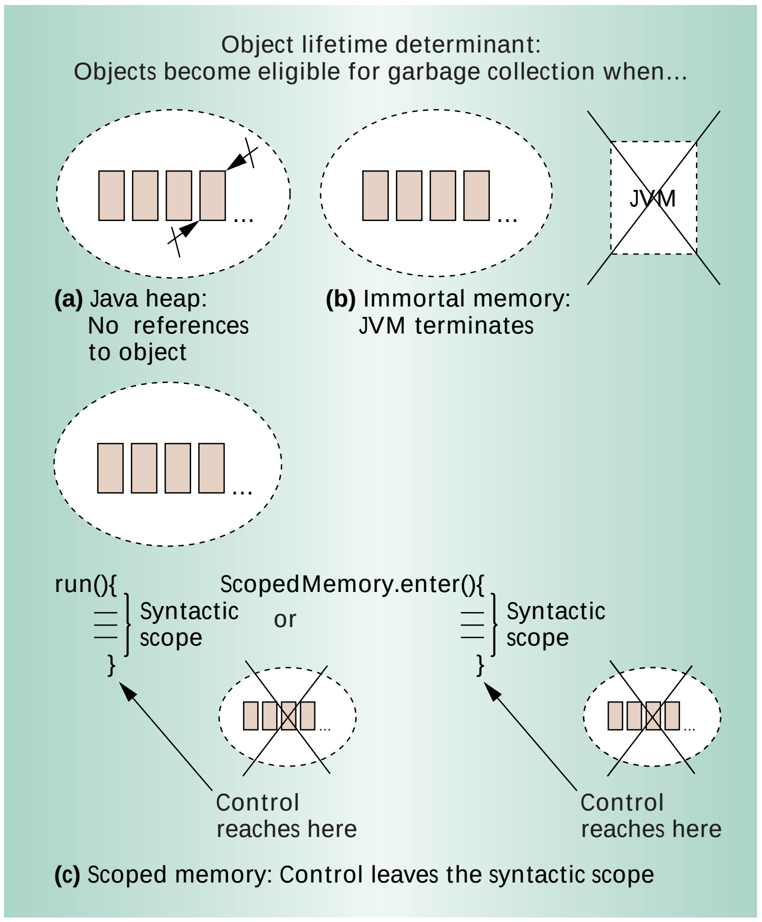
Object lifetime determinant:
Objects become eligible for garbage collection when...
JVM
(a) Java heap: No references to object
(b) Immortal memory: JVM terminates
run(){
Syntactic scope
}
or
ScopedMemory.enter(){
Syntactic scope
}
Control reaches here
Control reaches here
(c) Scoped memory: Control leaves the syntactic scope

Figure 1. How immortal and scoped memory differ from other methods of determining an object’s life. (a) The Java heap uses automatic allocation, in which the visibility determines the object’s life (if there are no references to the object, the system can deallocate it). Automatic allocation requires a garbage collector, however, which incurs delays. (b) In allocation using the RTSJ immortal memory, the object’s life ends only when the Java virtual machine (JVM) terminates. (c) RTSJ scoped memory uses syntactic scope, a special type of memory area outside the garbage-collected heap that lets you manage objects with well-defined lifetimes. When control reaches a certain point in the logic, the system destroys any objects within it and calls their finalizers.

SYNCHRONIZATION

In synchronization, the RTSJ uses “priority” somewhat more loosely than the conventional real-time literature. “Highest priority thread” merely indicates the most eligible thread—the thread that the scheduler would choose from among all the threads ready to run. It does not necessarily presume a strict priority-based dispatch mechanism.

Wait queues

The system must queue all threads waiting to acquire a resource in priority order. These resources include the processor as well as synchronized blocks. If the active scheduling policy permits threads with the same priority, the threads are queued first-in, first-out. Specifically, the system

- orders threads waiting to enter synchronized blocks in a priority queue;
- adds a blocked thread that becomes ready to run to the end of the run-ready queue for that priority;
- adds a thread whose priority is explicitly set by itself or another thread to the end of the run-ready queue for the new priority; and
- places a thread that performs a yield to the end of its priority queue.

Avoiding priority inversion

The synchronized primitive’s implementation must have a default behavior that ensures there is no

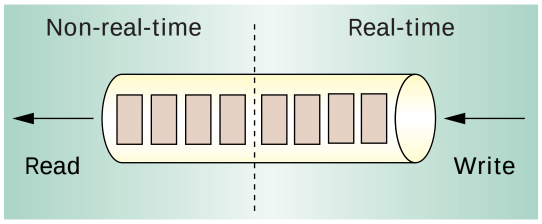

Figure 2. How threads in an RTSJ implementation communicate in a wait-free write queue. The unidirectional queue spans systems with different arrival dynamics (real-time to non-real-time).

unbounded priority inversion. This applies to conventional Java code if it runs within the overall RTSJ implementation as well as to real-time threads. The priority inheritance protocol—a well-known real-time scheduling algorithm[6]—must be implemented by default.

The specification also provides a mechanism by which you can override the default systemwide policy or control the policy to be used for a particular monitor, as long as the implementation supports that policy. The specification of monitor control policy is extensible, so future implementations can add mechanisms. A second policy, priority ceiling emulation (or highest locker), is also specified for systems that support it.[6]

Determinism

Conforming implementations must provide a fixed upper bound on the time required for the application code to enter a synchronized block for an unlocked monitor.

Sharing and communication among threads

Implementers are most likely to use a combination of regular Java threads, RTs, and NHRTs. The RTSJ permits locking between different thread types—even in the most contentious case, which is between regular threads and NHRTs. If an NHRT attempts to lock an object that either an RT or regular thread has already locked, priority inheritance happens as normal. There is one catch: The non-NHRT that has had its priority boosted cannot execute when the garbage collector is executing. Thus, if a garbage collection is in progress, the boosted thread is suspended until collection completes. This, of course, causes the NHRT to incur a delay because of the collection. To address this, the RTSJ provides mechanisms that allow NHRTs to communicate (a form of synchronization) with RTs and regular Java threads while avoiding garbage-collector-induced delays in the NHRTs.

The RTSJ provides queue classes for communication between NHRTs and regular Java threads. Figure 2 shows a wait-free write queue, which is unidirectional from real-time to non-real-time. NHRTs typically use the write (real-time) operation; regular threads typically use the read operation. The write side is nonblocking (any attempt to write to a full queue immediately returns a "false") and unsynchronized (if multiple NHRTs are allowed to write, they must synchronize themselves), so the NHRT will not incur delays from garbage collection. The read operation, on the other hand, is blocking (it will wait until there is data in the queue) and synchronized (it allows multiple readers). When an NHRT sends data to a regular Java thread, it uses the wait-free enqueue operation, and the regular thread uses a synchronized dequeue operation.

A read queue, which is unidirectional from non-real-time to real-time, works in the converse manner. Because the write is wait-free, the arrival dynamics are incompatible and data can be lost within it. To avoid delays in allocating memory elements, class constructors statically allocate all memory used for queue elements, giving the queue a finite limit.

If the regular thread is not removing elements from the queue at a high-enough rate, the queue may become full. The NHRT may not block on a full queue because the NHRT would incur delays from non-real-time activities such as garbage collection, and its execution behavior would then be harder to predict. Because of this restriction, a write may overwrite an existing element. This is reasonable because, given that the write cannot wait, no amount of buffering will ensure that all elements have a place in the queue. Such sharing is suitable for data that can be recovered, for data that higher level protocols will regenerate as appropriate, or for situations where data loss is expected and not a requirement for system correctness.

ASYNCHRONOUS EVENT HANDLING

The asynchronous event facility comprises two classes: AsyncEvent and AsyncEventHandler. An AsyncEvent object represents something that can happen—like a Posix signal or a hardware interrupt—or it represents a computed event—like an airplane entering a specified region. When one of these events occurs, indicated by the fire() method being called, the system schedules associated AsyncEventHandlers.

An AsyncEvent manages two things: the dispatching of handlers when the event is fired, and the set of handlers associated with the event. The application can query this set and add or remove handlers.

An AsyncEventHandler is a schedulable object, roughly similar to a thread. When the event fires, the system invokes run() methods of the associated handlers.

Unlike other runnable objects, however, an AsyncEventHandler has associated scheduling, release, and memory parameters that control the actual execution of the handler once it is fired. When an event is fired, the system executes the handlers asynchronously, scheduling them according to the parameters. The result is that the handler appears to have been assigned to its own thread. It may or may not actually be assigned to its own thread; the point is that it merely has to appear that way.

You can implement AsyncEventHandlers to use far fewer system resources than actual threads use. The system should be able to cope well even when there are tens of thousands of AsyncEvents and AsyncEventHandlers. The number of fired (in process) handlers should be smaller.

A specialized form of an AsyncEvent is the Timer object, which represents an event whose occurrence is driven by time. There are two forms of Timers: the OneShotTimer and the PeriodicTimer. OneShotTimers fire off once, at the specified time. If the current time is later than the time specified, the system fires the handlers immediately. PeriodicTimers fire off at the specified time, and then according to a specified interval.

The RTSJ represents clocks using the Clock class. Implementations may offer applications more than one clock. A special Clock object, Clock.get RealtimeClock(), represents the real-time clock and must be in all RTSJ implementations. Many objects that measure time can be instantiated with any of the Clock instances the implementation offers.

ASYNCHRONOUS CONTROL TRANSFER

Asynchronous control transfer lets you identify particular methods by declaring them to throw an AsynchronouslyInterruptedException (AIE). When such a method is running at the top of a thread's execution stack and the system calls java.lang. Thread.interrupt() on the thread, the method will immediately act as if the system had thrown an AIE. If the system calls an interrupt on a thread that is not executing such a method, the system will set the AIE to a pending state for the thread and will throw it the next time control passes to such a method, either by calling it or returning to it. The system also sets the AIE's state to pending while control is in, returns to, or enters synchronized blocks.

ASYNCHRONOUS THREAD TERMINATION

The system can use AIE directly or in combination with asynchronous events to implement asynchronous thread termination. Sometimes, by design, all the methods that an instance of RealtimeThread uses are declared to throw an AIE. In that case, when the system calls an interrupt() on the thread, the effect is similar to the deprecated Java stop() method. Unlike this method, however, the threads stop safely because code written under the assumption that it would not be interrupted or code in synchronized blocks completes normally.

PHYSICAL MEMORY ACCESS

The RTSJ defines two classes for programmers who want to access physical memory directly from Java code. The first class, RawMemoryAccess, defines methods that let you build an object representing a range of physical addresses and then access the physical memory with byte, word, long, and multiple-byte granularity. The RTSJ implies no semantics other than the set and get methods.

The second class, PhysicalMemory, lets you build a PhysicalMemoryArea object that represents a range of physical memory addresses where the system can locate Java objects. To construct a new Java object in a particular PhysicalMemory object, you can use either the newInstance() or newArray() methods.

The RTSJ strengthens the semantics of the scheduling, memory management, and synchronization algorithms and helps produce systems that are relatively accessible to temporal reasoning.

An instance of RawMemoryAccess models a raw storage area as a fixed-size sequence of bytes. Factory methods let you create RawMemoryAccess objects from memory at a particular address range or using a particular memory type. The implementation must provide and set a factory method that interprets these requests accordingly.

A full complement of get and set methods lets the system access the physical memory area's contents through offsets from the base—interpreted as byte, short, int, or long data values—and copy them to or from byte, short, int, or long arrays. By treating a value as an offset itself, a program can use a physical memory area to contain references to other data values in the same area. A program can also define a region of one memory area as a different memory area. The base address and size, and any offset into a physical memory area, are long (64-bit) values.

Relative to the Java Language and Java virtual machine specifications, the RTSJ strengthens the semantics of the scheduling, memory management, and synchronization algorithms. All the remaining areas except physical memory access help some classes of real-time systems and will produce systems that are relatively accessible to temporal reasoning. As a convenience, the RTSJ includes physical memory access features because many real-time systems require programmatic access to physical memory.

We chose to solve the problem of supporting real-time programming in Java by defining features and semantics that let programmers manage thread execution and reduce the overall unpredictability of execution for certain thread types. We could have focused on object attributes and required temporally predictable behavior defined by some set of attributes. Although this approach was interesting, we decided that, at least for now, the resulting programming paradigm was too far from what is now practiced. As the real-time software industry becomes comfortable with object-based programming, such approaches could become more viable.

The RTSJ is essentially complete. We may make minor changes as we code the reference implementation and test suites, but we expect all three components—reference implementation, test suites, and the RTSJ—to become final about the end of 2000. ✷

Acknowledgments

Many people helped the Real-Time for Java Experts Group, but we have space to mention only a few. We thank Rod Smith at IBM and Pat Sueltz formerly at IBM (now at Sun) for accepting and promoting IBM as the lead participant and Jim Mitchell and Ken Urquhart at Sun for creating and shepherding the Java Community Process. Among those at IBM who are building the reference implementation, Peter Haggar, Jim Mickleson, and David Wendt have provided valuable feedback. We thank the other members of the RTJEG primary team—Paul Bowman, Ben Brosgol, Peter Dibble, Steve Furr, David Hardin, Mark Turnbull—as well as the current and former members of the RTJEG consulting team: Rudy Beliardi, Alden Dima, E. Douglas Jensen, Ray Kaman, Alexander Katz, Masahiro Kuroda, Doug Locke, George Malek, Jean-Christophe Mielnik, Wolfgang Pieb, Russ Richards, Raj Rajkumar, Larry Rau, Mike Schuette, Simon Waddington, and Chris Yurkoski. And finally, thanks to all those who took the time to comment on the specification.

Text describing the RTSJ features and semantics is from RTSJ v0.8.1 by permission of Sun Microsystems.

References

1. The Real-Time for Java Experts Group, *The Real Time Specification for Java, Version* 0.8.1, 27 Sept. 1999; http://www.rtj.org/rtj.pdf.
2. *The Java Community Process Program Manual,* Sun Microsystems, Inc., Dec. 1998; http://java.sun.com/aboutJava/communityprocess/java_community_process.html.
3. J. Gosling, B. Joy, and G. Steele, *The Java Language Specification,* Addison-Wesley, Reading, Mass., 1996.
4. T. Lindholm and F. Yellin, *The Java Virtual Machine Specification,* 2nd ed., Addison Wesley Longman, Reading, Mass., 1999.
5. L. Sha, R. Rajkumar, and J. Lehoczky, "Real-Time Computing using Futurebus+," *IEEE Micro,* June 1991, pp. 30-33; 95-99.
6. L. Sha, R. Rajkumar, and J. Lehoczky, "Priority Inheritance Protocols: An Approach to Real-Time Synchronization," *IEEE Trans. Computers,* Sept. 1990, pp. 1175-1185.

Greg Bollella is a senior architect at IBM Corp. and lead engineer of the Real-Time for Java Experts Groups. Previously, he designed and implemented communication protocols for IBM. He holds a PhD in computer science from the University of North Carolina at Chapel Hill. His dissertation research is in real-time scheduling theory and real-time systems implementation. Contact him at bollella@us.ibm.com.

James Gosling is a fellow at Sun Microsystems and the originator of the Java programming language. His career in programming started by developing real-time software for scientific instrumentation. He has a BSc in computer science from the University of Calgary and an MSc and a PhD in computer science from Carnegie Mellon University.

The Inside Story

- Cutting Edge Stream Computing
- $100 Million in Sales & Growing
- Top Minds Creating Real-Time Solutions

Powerful, proven technology, leading-edge tools, professionals who are the best in the business – this is what's inside Mercury Computer Systems, Inc. Mercury's embedded, high-performance multicomputers transform unrelenting streams of digital data to information in real time for life- and mission-critical applications.

APPLICATIONS ENGINEERS
- **CA & VA:** DSP, C, UNIX, VME
- **MW or CH:** real-time systems, imaging

SOFTWARE ENGINEERS
- Real-Time OS
- Release/System Test
- Tools/Algorithms
- Wireless

PRODUCT MARKETING MANAGER

HARDWARE ENGINEERS
- **Board Designers:** FPGA, PCI, PowerPC, PowerQUICC
- **ASIC Engineer:** signal integrity, C, UNIX, ASIC/FPGA, VHDL
- **Diagnostics Engineer:** C, ClearCase, PowerPC

SYSTEM ENGINEERS
- **SIGINT/IMINT:** DSP/IP, SAR, ATR
- **Embedded DSP:** real time computing, VME, PCI

Join a talented, motivated team in an exciting, fast-paced environment that encourages creativity and work/life balance. Mercury offers fully competitive salaries and benefits, state-of-the-art facilities and strong revenue growth. Equal opportunity employee.

FOR AN INSIDE OPPORTUNITY, SEND YOUR RESUME TO:

Email: mercury@webhire.com in ASCII, no attachments; Subject line: (Source Code IEEE600)
Mail: Mercury Computer Systems, Inc., Recruiting Department
199 Riverneck Road, Chelmsford, MA 01824-2820
Fax: (978) 256-1746

www.mc.com/jobs

MERCURY Computer Systems, Inc.