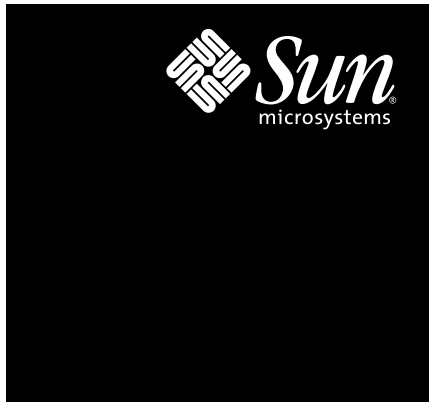
The Real-Time Java Platform



A Technical White Paper
June 2004

Project Mackinac, based on the Real-time Specification for Java (JSR-001) and Sun's Java HotSpot™ technology, anticipates the needs of the real-time and embedded developer community

Please
Recycle

Table of Contents

Real-time System Trends 1
More Software and More Complex Applications 1
More Interaction with the Physical World 2
Increasing Interconnectedness and Hierarchical Networks 3
Real-time and Non Real-time Coexisting on a Single Node 5
Stochastic versus Deterministic Systems 6
Implications for Real-time Systems 8
Java as a Real-time Platform 8
Why RTSJ? 9
Technological Advances in the RTSJ 10
Scheduling 10
Memory Management 12
Synchronization 14
Asynchronous Event Handling 15
Asynchronous Transfer of Control 17
Physical Memory Access 18
Mackinac Performance 18
Throughput 18
Latency and Jitter 19
Conclusion 20

Abstract

This white paper introduces Project Mackinac, the first commercial implementation by Sun Microsystems of JSR-1, the Real-Time Specification for Java. The Mackinac product supports both hard real-time and non-real-time functionality in a single system based on the Java Hotspot platform, and delivers performance that is competitive with compiled, real-time solutions, such as C and C++.

Real-time System Trends

Like all sectors of the computing industry, real-time computing is changing rapidly. These changes are going to have a profound effect on the real-time community. Among other things, they create a demand for more real-time programmers while simultaneously challenging some of our basic assumptions about what a real-time system is.

The changing quality of real-time systems will require real-time developers to consider factors heretofore of interest only to developers of business systems. In the longer term, it is likely that some of the concepts of real-time system design will migrate in the opposite direction as well, so that the enterprise application developer of 2020 might routinely manipulate program elements that are, today, virtually unknown outside the real-time developer community.

We believe there are four major trends in the real-time developer community:

More Software and More Complex Applications

The first trend is toward having more and more of the functionality of embedded devices defined in software and less in hardware. Figure 1 illustrates this trend.

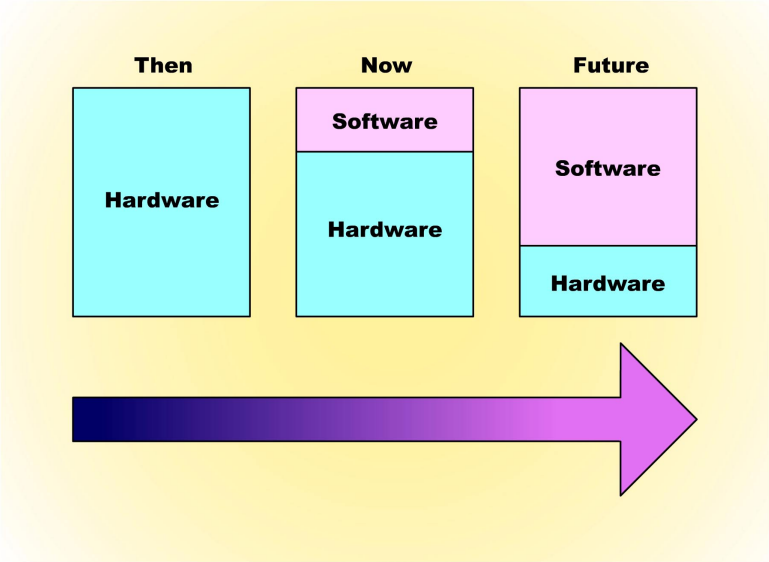


Figure 1. Changing Mix of Hardware and Software in Real-Time Systems

The first real-time systems were implemented entirely in hardware. As digital devices replaced the original analog ones, developers began to implement more and more features in software. Today, there is a pronounced trend toward real-time systems in which more and more of the functionality is provided by software running on commodity microprocessors, rather than on logic implemented in hardware.

One reason for this trend is that it is faster, easier, and more cost-effective to implement designs in software than in hardware. By moving to a software-intensive solution, manufacturers can reap valuable time-to-market and cost-savings benefits.

Another reason has to do with market forces. Consumers are demanding more and more sophisticated features in everything they buy, leading competitors to pack ever more functionality into their products. To see this trend in action, you have only to compare this year's cell phones and digital cameras to those available just a couple of years ago. Every embedded device is following this trend, without exception. Consumer electronics, such as DVD and MP3 players, cell phones, and digital cameras, are common examples of small, embedded systems, but it is just as valid to view automobiles, aircraft, warships, and exploratory spacecraft as embedded systems, too. The same trends apply to all embedded systems, small and large.

As real-time systems grow more and more complex, the high-touch, low-level programming methodology that served real-time developers well in the past will no longer suffice. Low-level programming will become uncompetitive from a cost standpoint, and will not be able to provide the level of abstraction and data modeling required for enterprise-level software applications. To meet these needs, real-time systems will have to provide higher-level abstractions and advanced tools for functional and temporal programming.

More Interaction with the Physical World

Greg Papadopoulos, Executive Vice President and Chief Technology Officer of Sun Microsystems, prepared the slide shown in Figure 2 to illustrate some fundamental changes in the general computing industry over time. Although he intended the slide to illustrate how the proliferation of smaller devices will drive volume increases across the computing industry, the slide also illustrates a point that is germane to the present discussion: many of the new devices that will come into use in the near future are ones that interact with the physical world in some way. This trend will greatly increase the number of real-time embedded systems in use.

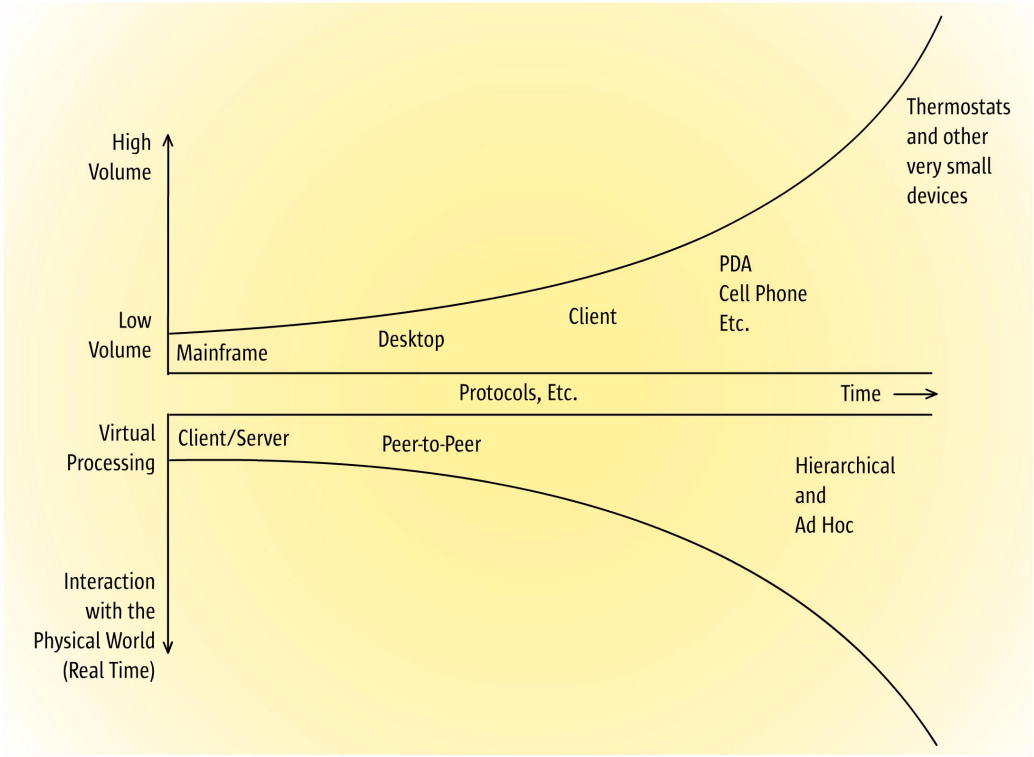


Figure 2. Trends in Computing

The second trend is, therefore, toward more interaction with the physical world. All this new real-time functionality will require a lot of real-time programming, and that means more real-time programmers will have to be trained. At the same time, many of these new devices will be commodity items – things like smart thermostats and light switches – which means that there will be considerable pressure for the cost of real-time development to come down.

This trend indicates a need for an easy-to-use real-time programming environment and a large pool of experienced developers capable of designing, programming, and testing real-time systems – two areas in which the Java programming language offers compelling advantages.

Increasing Interconnectedness and Hierarchical Networks

More and more of the new real-time devices manufactured over the next few years will communicate with other devices over networks of various kinds. Toward the left side of Figure 2, the prevalent computational model is client-server, which is straightforward and simple. But over time, more and more devices tend to become connected in a hierarchical or *ad hoc* manner, which drives up the complexity of the software, making the work of the designers much harder, and requiring more sophisticated programming languages to do the job correctly and efficiently.

Figure 3 shows a representative embedded system, which is a collection of sensors, actuators, and a control system.

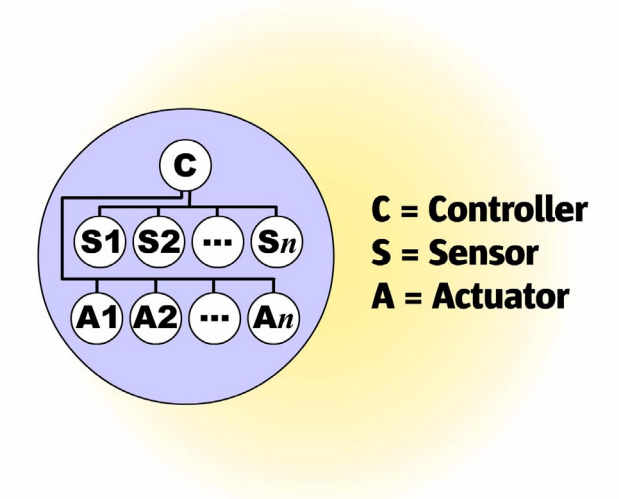


Figure 3. The Basic Imbedded System

The industrial control world involves the use of lots of sensors, generating lots of data. Historically, most of this data has been discarded. For example, temperature changes actuate a thermostat, which then turns the heat on or off, but the temperature data points and event history are not usually stored for later use.

Even in a modern, computerized embedded device, such as an automobile, much of the collected data is discarded. A subset of the original data is often stored in a log for use by service technicians, but even the data in these logs is often discarded after the service call to make room for new data. However, the trend is for embedded devices to evolve not as standalone objects, but as systems of interconnected subsystems, each of which is, in turn, a collection of interconnected embedded devices, as shown in Figure 4

.

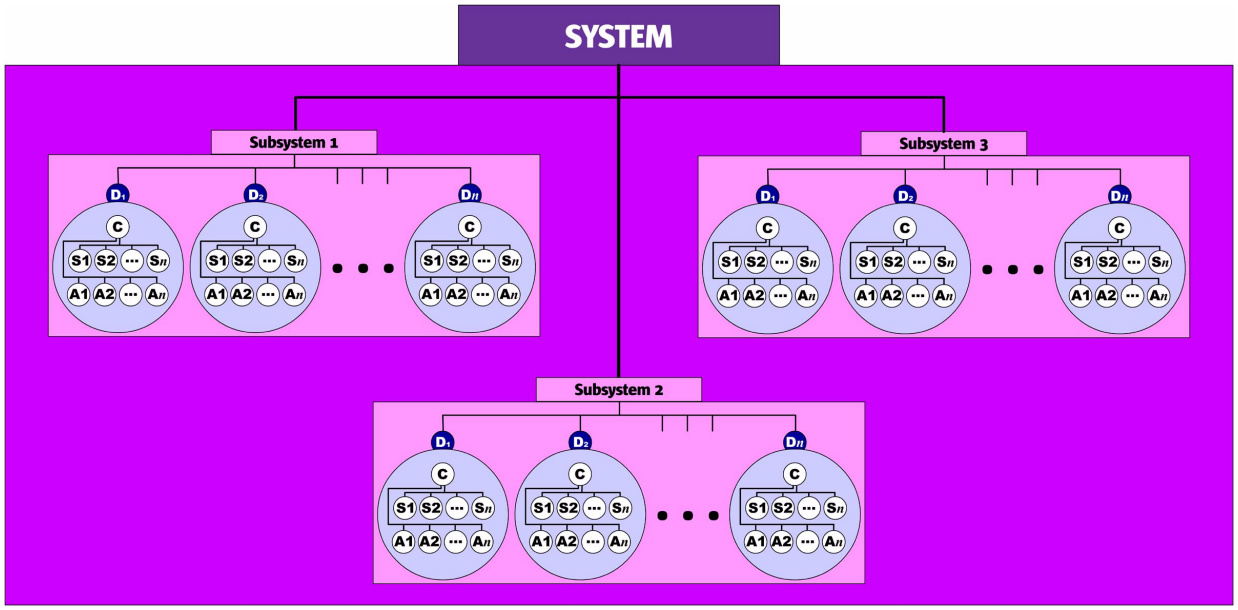


Figure 4. A Hierarchical Meta-Control System (HMC)

We are seeing a strong desire among the manufacturers of embedded devices to collect them into a hierarchy of control nodes that culminates in some kind of control center. That control center potentially has to manage not only individual embedded devices, but also to perform some or all of the following tasks as well:

- Controlling components comprising several embedded devices
- Controlling subsystems comprising many components
- Controlling systems comprising many subsystems
- Interfacing with super-systems comprising many systems

Hierarchical systems of this kind are naturally going to require a very different approach to the management of data. There will be a need to collect data into control hierarchies to facilitate higher degrees of control over industrial devices because industrial devices these days are very complex. This will require us to expand the notion of what a real-time node is.

At present, we tend to think only of the small, embedded device, but a fully developed system of the general type described above might include, at the higher levels of the hierarchy, powerful enterprise-level computing resources. We might refer to systems such as these as *Hierarchical Meta-Control* (HMC) systems. In complex, hierarchical control systems like these, some processes need to be managed using real-time systems while others do not. But as we start to explore what such complex hierarchical systems should look like, we're finding that we often need real-time capabilities at *each* level of the hierarchy. That means we need a real-time architecture that spans a range of hardware – one that supports the embedded microprocessor, the desktop computer, the individual workstation, and the distributed enterprise computing system.

The larger trend is one of *control system proliferation*, in which control systems need to be able to interact with subsystems as well as with super-systems, and with peer systems that may or may not belong to the same enterprise.

All of these factors combine to require a change in the programming models currently being used. The low-level languages currently used to program real-time systems are not up to the task of developing large, distributed systems that span a range of complexity from board-level systems to large enterprise systems. Only a high-level, high abstraction, modern, real-time, well-supported language and runtime can make this range of device, application, and infrastructure complexity manageable for the architects and developers who are designing complex real-time systems.

Projects with extended lifetimes, such as major weapons systems that must continue to be supported and upgraded for decades, also require a vendor-neutral, real-time platform and programming language that system developers and architects can count on to be there over the long haul. Once again, the Java platform, with its community-based specification model and wide range of vendor choices, is well positioned to meet this industry need.

Real-time and Non-real-time Coexisting on a Single Node

As hierarchical real-time systems evolve, it is almost certain that desktop computers, servers, and enterprise computers will be used to perform control functions for hierarchical systems, while simultaneously interacting with business systems that are not real-time systems. With respect to real-time systems, one set of problems arises simply as a result of co-locating a real-time process and a non-real-time one. A second problem arises to the extent that data is required to flow between real-time and non-real-time systems.

The Real-time Specification for Java (RTSJ), located at http://www.jcp.org/en/jsr/detail?id=1, was created with this future trend in mind, and one of its most important features is that it supports different levels of real-time behavior, as shown in Figure 5.

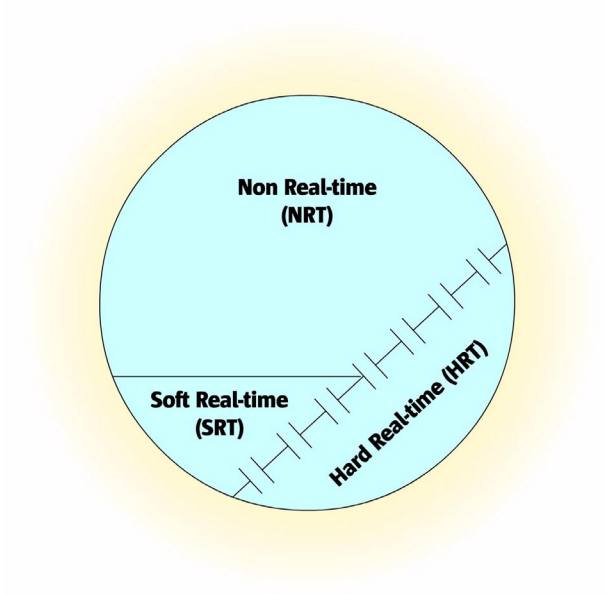


Figure 5. How RTSJ Supports Different Levels of Real-Time Behavior

Note the broken line separating the Hard Real-time (HRT) section from the rest of the system. This broken line represents specialized data transfer queues that allow data to flow to and from the HRT system in a predictable and well-defined way so that the developer can make appropriate trade-offs between blocking, on the one hand, and data loss on the other, based on the predictable and well-defined RTSJ system. These trade-offs are discussed in more detail in the next section.

Stochastic versus Deterministic Systems

The RTSJ describes a system that permits deterministic (real-time) processes to co-exist on a single system with stochastic (non-real-time) processes. This is not a trivial undertaking, as you will see in a moment.

Suppose you have a stochastic process and a deterministic one, each with a nominal computational cycle of *t*. Figure 6 shows the likelihood that each of these two processes will be complete at any moment in time. A stochastic process is very likely to be complete at or shortly after the minimum time required it takes for that particular computer to execute the associated code segment, but the actual time varies with load, so the probability tails over time but never theoretically reaches zero. By contrast, a deterministic process with a computational cycle of *t* will be completed precisely at time *t*.

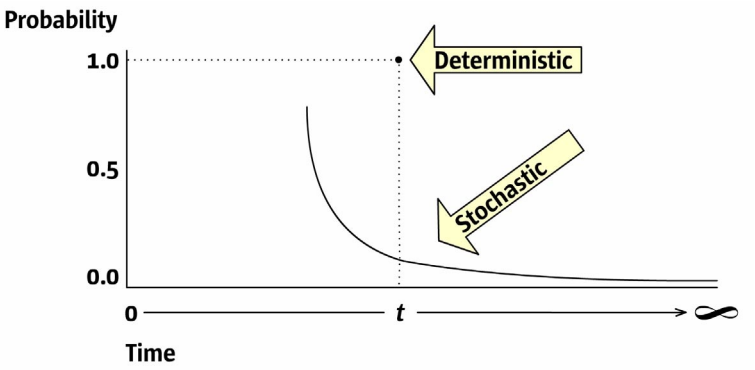


Figure 6. Stochastic vs. Deterministic Behavior

The real-time world is deterministic. The non-real-time world is stochastic. If you attempt to share data between these two worlds in the standard computer science way, you run into a problem. This was the working definition of a real-time programming environment used by the developers of the RTSJ:

> "The programming environment must provide abstractions necessary to allow developers to correctly reason about the temporal behavior of application logic. It is not necessarily fast, small, or exclusively for industrial control. It is all about the predictability of the execution of application logic with respect to time."

Put another way, a hard real-time system must always strive to eliminate conditions that reduce its ability to guarantee predictable execution of application logic with respect to time.

The standard computer science way to transfer data between two processes is to write the data into a queue, as shown in Figure 7. In the situation illustrated here, a deterministic process generates a data object every 10 ms while a stochastic process consumes those data objects approximately every 10 ms. The problem arises because the rate at which the stochastic process completes its computational cycle is only approximate; any given computational cycle might take less than or longer than 10 ms.

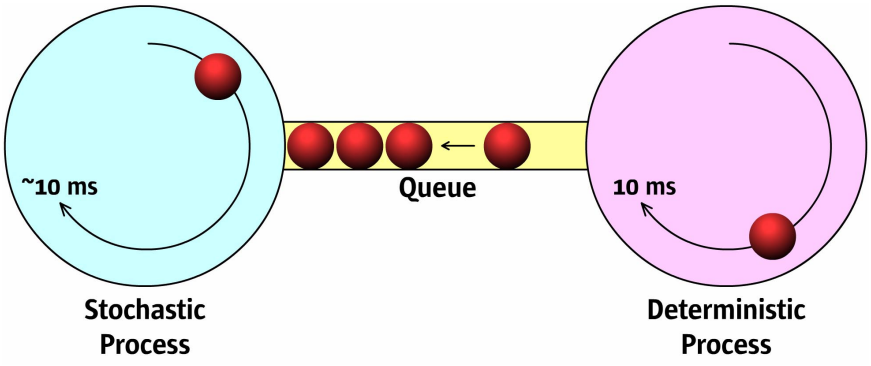


Figure 7. The Queuing Problem

Therefore, it is possible that the stochastic process will be late enough times to overflow even a very large queue. To cover that possibility, the theory states that, for approximately equal rates of data production and consumption, the data queue in Figure 7 must be of *infinite length* to ensure that the writer never gets blocked and no data ever gets lost. Because the practical reality is that a data queue cannot be infinitely long, the problem is theoretically unsolvable.

This theoretical problem has the practical effect of forcing the application developer to make a choice between losing data or turning the deterministic (real-time) process into a stochastic (non-real-time) process. Why? When the queue fills up, the process on the right – unless it is designed to discard data under such circumstances – must wait for a space in the queue to open up before it can place its data there (this waiting for a filled queue to open up is called *blocking*). Blocking changes the computational cycle of the supposedly deterministic process from exactly 10 ms to some other value, and when that happens, the process ceases to be deterministic and becomes, in effect, stochastic.

Implications for Real-time Systems

The systems of one environmental controls market leader generate three petabytes of information each week, collected from sensors installed in the buildings it monitors. Most of that information is discarded. This Sun customer – and doubtless many others in similar circumstances – would like to put the data they collect to better use, but cannot do so without an HMC system. Unfortunately, no systematically developed HMC currently exists.

To make it possible to monitor, control, and derive optimal value from the information generated by complex hierarchical systems, system developers will need an infrastructure that allows them to build and interconnect a wide range of systems:

- Individual real-time devices
- Small networks of real-time devices
- Large networks of real-time devices managed by HMC systems
- Very large distributed systems that combine lots of computational power with varying levels of real-time support, including simultaneous support of multiple HMCs

Designing the logic to run such complex real-time systems is simply infeasible using the low-level hardware programming techniques real-time system developers have traditionally relied on. An entirely new infrastructure is needed – one that provides the following elements:

- A high-level programming language
- Advanced Program Development Methodologies (APDM) for embedded/small devices
- APDM for control nodes
- Feasibility testing tools
- Protocols that are compatible with real-time needs
- A comprehensive architecture that supports chip- and board-level devices as well as enterprise-level real-time, HMCs, and mixed-use systems.

This white paper focuses primarily on the first item: the high-level programming language for next-generation real-time development projects.

Java as a Real-time Platform

The Real-time Specification for Java (RTSJ) and Sun's implementation of that specification are, of course, extensions of the Java programming language and Java Virtual Machine (JVM). Before delving into the details of how the RTSJ and Mackinac handle specific real-time tasks, it might be a good idea to pause and ask a prior question, namely, what makes the Java language suitable for real-time work in the first place?

There are three commercially viable choices for real-time software development:

- C
- C++
- Java

While the C programming language is currently quite popular for small, embedded devices, the limitations of C quickly become apparent as project complexity increases. It is safe to say that C does not provide enough abstraction to program HMC systems effectively, and is quite cumbersome even on midsize projects. So while it is feasible to program individual embedded devices in C, this approach turns out to be a dead end when viewed in the context of the trends we identified earlier.

By contrast, C++ is complex and cumbersome; its popularity is already in decline from its heyday ten to fifteen years ago. Putting aside its complexity, the C++ approach simply cannot guarantee the long-term support, product development, and pool of qualified programmers that will be critical to the successful development of very large real-time systems and development projects that span decades, as NASA and military projects do.

The Real-Time Java platform is a better choice on all counts. Java offers more in the way of abstraction than C, avoids the complexity of C++, and is embraced by a large and growing community of active software developers. The things that make Java ideal are:

- High programmer productivity
- High industry acceptance
- Very high levels of support from many vendors
- A well thought-out evolutionary path administered by a very effective open-systems organization, the Java Community Process (JCP)

What about efficiency? Java is often viewed in the embedded world as being inefficient – which is to say, that it takes more CPU cycles to do the same job in Java than in C or C++. The question of the comparative efficiency of Java relative to C and C++ is a complex one, so it's worth taking a few moments here to explore it.

The first thing to be noted is that compiled Java does indeed take more CPU cycles to execute the average line of code than does C, and about the same as C++. So on the face of it, the question becomes this: As the trends we noted earlier unfold, will that extra overhead be justified to get the other benefits Java offers? For larger systems, such as the ones we characterized earlier as Hierarchical Meta-Control (HMC) systems, the answer is almost certainly yes.

But this still leaves the question of less complex applications. Even if Java were to become the de facto standard for large and very large real-time systems, would not C still remain a better choice for smaller embedded systems? As it turns out, the answer you get depends on how you ask the question because there are different ways to measure language efficiency.

Our calculations and benchmarks suggest that a well-implemented real-time Java system should have about the same throughput as a well-implemented real-time C++ system.

Why RTSJ?

Based on all of the considerations discussed above, Java would appear to be a very good candidate for real-time development, if it were not for one glaring problem: Java is not a real-time system. The fact was not lost on the early Java community, and in fact the very first Java Specification Request (JSR) issued by the JCP on 15 December 1998 was "JSR 1: the Real-time Specification for Java."

Mackinac is Sun's implementation of that specification; its purpose is to provide a real-time implementation that meets the stringent needs of real-time developers while continuing to offer all of the other advantages of the Java programming language, which are described in more detail below.

The United States Department of Defense currently estimates that a software project's probability of failure, if the software system is characterized by distributed real-time requirements, is 80%. That is to say, 80% of such projects either fail to reach completion or are useless when delivered.

The question is, will RTSJ allow such large-scale real-time development efforts to succeed in the future? We believe that RTSJ has a better chance of success than any other conceivable option, for two reasons:

- The RTSJ specification is itself a remarkable achievement, and is arguably the most successful attempt ever made to grapple with current and future real-time issues. The next section will discuss in detail how the RTSJ handled these technical challenges.
- The RTSJ specification extends the Java specification, and the Java language is supported by the Java Community Process, the most inclusive, far-reaching, and successful collaborative software language consortium in history. The Java language's industry acceptance is unexcelled; its cross-platform compatibility legendary, and skilled Java programmers are abundant.

While the RTSJ provides for the support of board-level embedded systems as well as mid-range and enterprise-level ones, it allows implementers to choose the area or areas in which they want to provide solutions. In Project Mackinac, Sun chose to address the needs of real-time systems that are integrated as part of a larger system solution. The targeted uses include monitoring, control, and management on SPARC/x86 Sun computers running Solaris.

The initial release of Project Mackinac will address the needs of the midrange development project (desktop- or workstation-based). Future releases will include support for very small devices, as well as support for enterprise-level real-time applications.

Technological Advances in the RTSJ

The RTSJ makes several modifications to the Java specification in order to make true real-time processing possible. In particular, the RTSJ represents important technological advances in the following five areas:

- Scheduling
- Memory Management
- Synchronization
- Asynchronous Event Mechanism
- Asynchronous Transfer of Control

A sixth item is one that real-time developers expect and demand in a real-time system:

- Physical Memory Access

The following subsections discuss each of these six topics in turn.

Scheduling

Strictly speaking, *scheduling* is the ordering of all tasks to be performed by the real-time system, while *dispatching* is the system the operating system uses to decide which task to run next. However, in keeping with common usage, the RTSJ treats both together under the combined heading of scheduling.

A real-time application consists of one or more tasks that can be described in terms of the following parameters:

$T_1, T_2, \ldots, T_n$– Tasks to be performed in the real time system
$C_1, C_2, \ldots, C_n$– Cost of each task (how long it takes to run each task)
$R_1, R_2, \ldots, R_n$– Release time for each task (time that task becomes available to run)
$D_1, D_2, \ldots, D_n$– Deadline for each task (when each task needs to be complete)

Figure 8 is a typical scheduling diagram, showing tasks T_i, T_j, and T_k and their relative execution durations (costs).

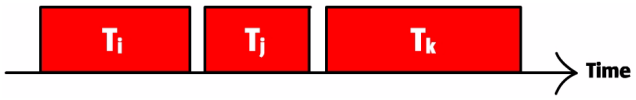


Figure 8. Scheduling

For any *n* tasks, there are *n*! possible orders in which they might be executed, and many more than that, if tasks can be interrupted and executed in pieces. Each ordering of the *n* tasks represents a schedule. A schedule is termed *feasible* if it meets all of the temporal conditions of all of the tasks or i*nfeasible* if it does not. Specifically, a feasible schedule is one in which each task T can be released at its release time R, get at least C units of execution time, and be complete before its deadline D.

The RTSJ says that programmers who have programs for which they want to meet temporal conditions should use some sort of analysis to ensure that the system can meet its deadlines. However, the RTSJ leaves it up to the implementer to provide feasibility testing and does not require feasibility testing in the base implementation. It stipulates that if no feasibility test is included, the RTSJ implementation shall return true whenever the test method is invoked.

The latitude on the part of the RTSJ is so that implementers can choose a feasibility-testing algorithm that best suits the real-time needs of the application types for which they are designing their RTSJ implementation.

Project Mackinac will eventually include a feasibility algorithm that is tailored to the needs of Sun's initial customers. As time goes on, there might be different versions of Mackinac, each of which tests feasibility in a different way to suit the special needs of a specific real-time developer community.

The RTSJ defines three types of processes and stipulates that all implementations shall, at a minimum, provide for handling all three types. The three types of processes are:

Periodic – A periodic task has a hard deadline and a fixed period; it is released at regular intervals. The developer defines, once, a cost C, period P, and a deadline D, and when the periodic task runs, it gets C units of execution time and a deadline D once every period P.

Sporadic – A sporadic task has a hard deadline but no fixed period; it is released by an event that does not occur at regular intervals, and it has a minimum interarrival time (the interval between the start of one recurring process and the start of the next one). The developer defines, once, the triggering event, a cost C, a deadline D, and sets aside enough time in each period for the execution of the sporadic process, should it need to execute then. The sporadic process gets C units of execution time and a deadline D every time the triggering event occurs, to be executed as soon as the time set aside for executing that sporadic process comes around. By deferring the execution of sporadic processes until a scheduled start time, they can be handled as periodic processes, thereby eliminating timing problems.

Aperiodic – An aperiodic task is released by an event that does not occur at regular intervals, and has no minimum interarrival interval. Because aperiodic tasks are so difficult for real-time systems to handle, developers will be interested to know how Mackinac will handle them. Mackinac does this using a sporadic process, known as the *sporadic server*, to service all aperiodic events that have occurred since the last time the sporadic server ran.

Aperiodic processes do not have fixed deadlines, and they can occur with arbitrary frequency. The way Mackinac prevents aperiodic processes from disrupting the temporal behavior of the system is by limiting the execution time for *all* aperiodic processes to the time set aside for the running of the sporadic server;, at the end of which time the aperiodic processes are suspended until the next time the sporadic server runs, when they resume processing from the point at which they were suspended.

There is another factor that has serious practical ramifications for real-time schedulers and dispatchers: how to handle tasks that fail to complete before their scheduled deadlines. A periodic task is defined by two values, C and

P, as shown in Figure 9. When there are many periodic tasks executing with various periods and costs, the feasibility analysis must cover the whole set to see if all tasks will meet their deadlines. In order for the feasibility analysis to do this reliably, the real-time system must perform *cost enforcement*, meaning that the two values, C and P, are treated as a contract.

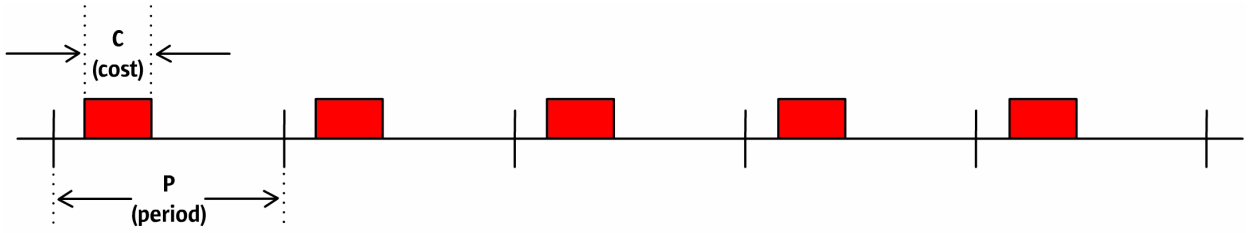


Figure 9. Cost and Period

Version 10 of the Solaris platform will include support for cost enforcement in the dispatcher code. If a thread's contract is for a C of 1 and a P of 5, the system enforces that contract. If the task is not complete by its scheduled deadline, the system will stop the task and not resume the task until its release time comes around again, in the next period. The effect is to punish only the process that overran its deadline, while preserving the temporal integrity of the rest of the real-time system.

The RTSJ allows the developer the option of writing cost overrun handler (COH) and missed deadline handler (MDH) logic for each process. These are asynchronous event handlers that get scheduled for execution when either of those two events happens. If in any period P a task attempts to use more than the cost C stipulated in its contract, the task is stopped and its COH executed. Similarly, when a task misses a deadline, its MDH is executed.

There are two predominant scheduling algorithms used in real-time systems, each of which has advantages.

Earliest Deadline First (EDF) – EDF is a dynamic scheduling principle that queues processes based on priority. As the current process ends, it is placed at the end of the queue. The system then searches the queue process that has the most imminent deadline and schedules that process for execution next. EDF generally provides better CPU utilization than static scheduling techniques. The major disadvantage of EDF is that the system becomes unpredictable as soon as processes begin to miss their deadlines.

Rate Monotonic Scheduling (RMS) – RMS is a static scheduling principle that can be used when the processes to be scheduled meet certain criteria. RMS assigns priority based on the execution period of each process, as follows: the highest priority goes to the process with the shortest execution period, the next highest priority goes to the process with the next shortest execution period, and so forth. The advantage of RMS is that it is possible to guarantee that all processes will meet their deadlines. The disadvantage is that CPU utilization is generally in the range of 70%.

The selection of a scheduling algorithm depends on factors that include the nature of the real-time application and the requirements of the system's feasibility analysis scheme. Project Mackinac will consider all relevant factors when choosing the scheduling model for specific product releases.

Memory Management

The trends in real-time development suggest that in the future, real-time applications will, in fact, become mixed applications, with various threads running in the hard real-time (HRT), soft real-time (SRT), and non-real-time (NRT) zones, depending on the degree of temporal control needed in each case.

In the C / C++ world, memory management is done manually; program logic determines when memory is allocated (malloc) and when it is freed (free). Program logic controls the lifetime of a memory object. In Java, by contrast, memory management is automatic. Program logic still decides when objects are created in memory (new),

but there is no way for the programmer to exercise direct control over the freeing of that memory. The Java garbage collector handles this task on behalf of the application.

Developers of non-real-time applications derive a great benefit from the garbage collector, which saves programming time and makes applications more reliable. However, the garbage collector is not compatible with HRT applications because it is difficult to predict when garbage collection will occur and how long it will take to complete.

The RTSJ provides the programmer with an environment in which application logic suffers zero interference from the garbage collector, by introducing a new memory model called *scoped memory*. In scoped memory, the lifetime of a memory object is determined by program scope, as shown here:

```
run() {
    //This is the scope of this thread.
}
```

The RTSJ allows the application developer to assign everything in the run() method, including calls to other methods, to an application-defined heap. Application-defined heaps are accessed by logic in a particular scope. So when you create a thread, you can give it one of these application-defined heaps. The garbage collector never collects objects in these heaps. Anytime a new() occurs in this scope, the object goes into the application-defined heap, not into the regular Java heap. The entire contents of the application-defined heap goes out of scope as soon as the application completes the run() method. As soon as it goes out of scope, all of the memory objects in the application-defined heap are destroyed instantaneously.

A garbage collection process takes time because the garbage collector has to traverse the entire heap, figure out which objects are still pointed to, and destroy only those objects that are not pointed to. By contrast, there is no time lost when the application-defined heap is cleared. Moreover, the garbage collector halts execution of all its associated threads while garbage collection is going on. Threads that use scoped memory suffer zero interference from the garbage collector.

Another memory model unique to the RTSJ is *immortal memory*. Like scoped memory, immortal memory in never subject to garbage collection; unlike scoped memory, however, memory objects created in immortal memory remain in memory for the duration of the application, even if there are no references to them.

Both scoped and immortal memory are reserved for HRT threads. The RTSJ provides for SRT threads as well. What makes SRT “soft real-time” is that threads in this zone do take advantage of automatic garbage collection. However, the SRT garbage collector must be able to interact with the application in order to avoid interfering with the predictability of the real-time applications running in the SRT zone. The initial release of Project Mackinac might support SRT.

Synchronization

The problem RTSJ needs to address is that of unbounded priority inversion. Priority inversion occurs when a high-priority task cannot complete execution because a lower-priority task has a lock on a resource needed by the higher-priority task. Bounded priority inversion, shown in Figure 10, does not represent an insurmountable problem, because the delay in completing the high-priority task due to priority inversion is bounded by the duration of the lower-priority task that must release the locked resource, and both are deterministic.

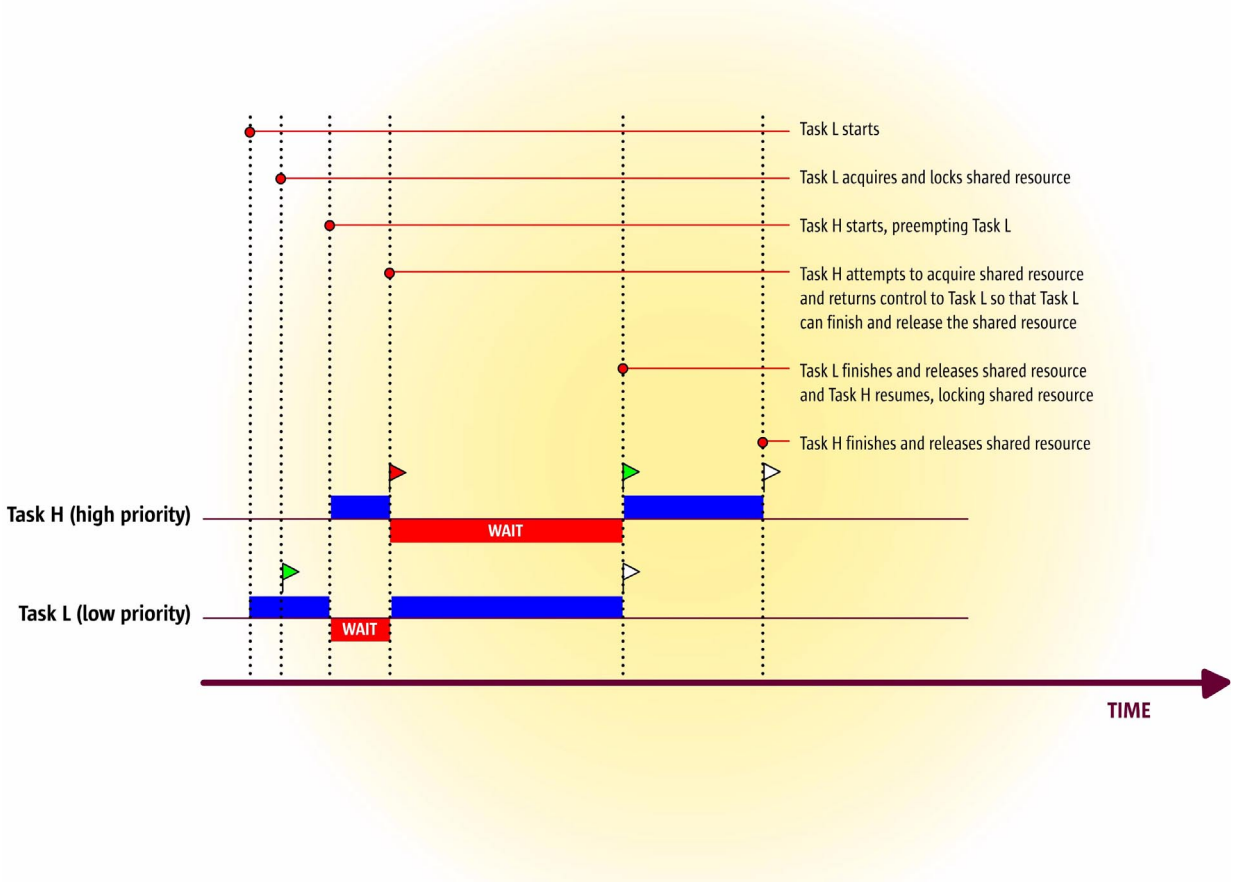


Figure 10. Bounded Priority Inversion Timeline

Unbounded priority inversion, shown in Figure 11, is a much more serious problem, because it is possible for an arbitrary number of tasks having intermediate priority levels to delay the completion of the low-priority task in unpredictable ways.

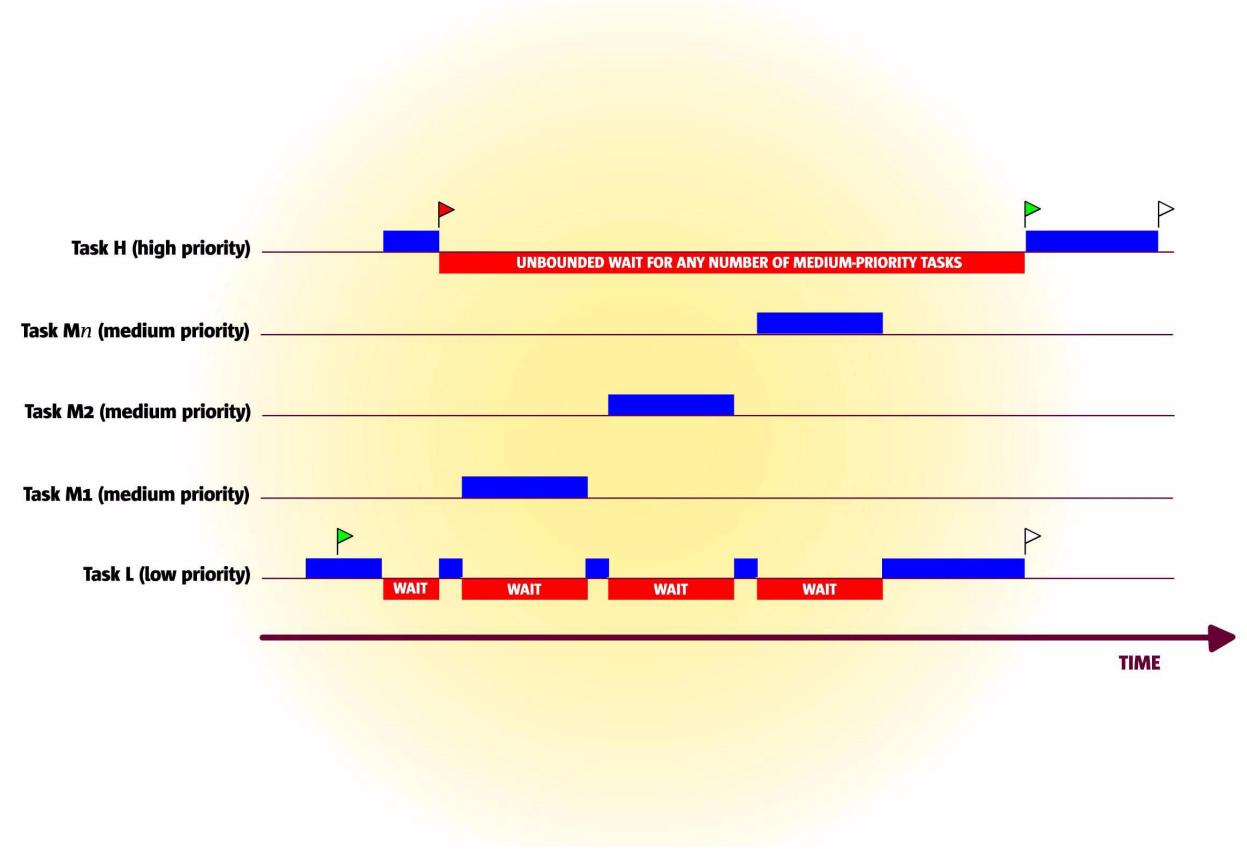


Figure 11. Unbounded Priority Inversion Timeline

There are three algorithms commonly used to avoid unbounded priority inversion, but they share one common object: to boost the priority of a low-priority task that has locked a resource needed by a high-priority task, which avoids preemption by intermediate-priority tasks. The RTSJ stipulates that implementers must apply a priority inheritance protocol to the Java keyword *synchronized* to avoid unbounded priority inversion.

Asynchronous Event Handling

The RTSJ provides several facilities for arranging asynchronous control of execution, some of which apply to threads in general while others apply only to real-time threads. These facilities fall into two main categories: asynchronous event handling, described in this section, and asynchronous transfer of control (ATC), which includes thread termination, described in the next section.

The RTSJ defines two classes of objects for implementing asynchronous event handling:

- An object called AsyncEvent (something that can happen)
- An object called AsyncEventHandler (logic that the developer writes, to be executed when that event happens)

An AsyncEvent can occur in either of two ways. Figure 12 shows the first of these, in which AsyncEvent is *fired* by program logic calling AsyncEvent.fire().

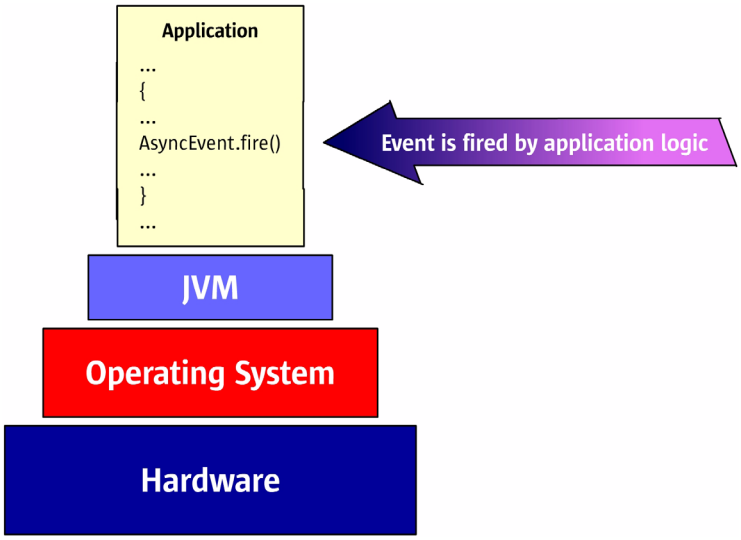


Figure 12. Firing an AsyncEvent

Figure 13 shows the second way in which an AsyncEvent can be triggered, by being bound to a *happening*.

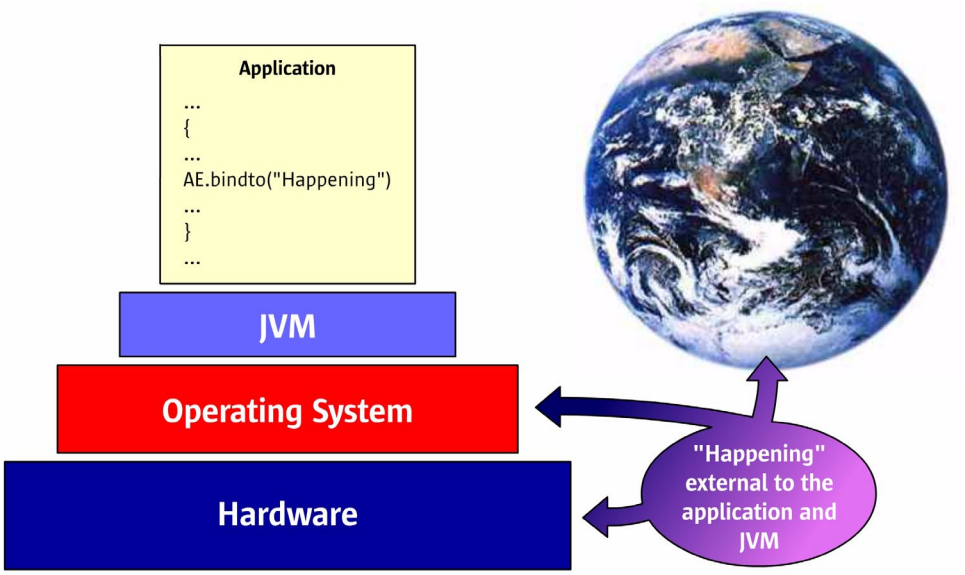


Figure 13. AsyncEvent Bound to "Happening"

The names of happenings, except for a few well-known ones, are defined by the RTSJ implementation. The programmer binds an AsyncEvent to a happening. For example, the AsyncEvent might be bound to an event in the hardware. If it is, then when that event occurs in the hardware, the AsyncEventHandler gets scheduled for execution just as if the program logic had called AsyncEvent.fire().

When the AsyncEvent occurs, all AsyncEventHandlers that are bound to that AsyncEvent are scheduled. AsyncEvent and AsyncEventHandler are indistinguishable from threads with respect to scheduling and dispatch, except that they do not occur until an event occurs.

AsyncEvents and AsyncEventHandlers are unique to RTSJ. The RTSJ uses them to decouple the occurrence of an event from the execution of the response to that event. It is this decoupling that gives the developer better control over the temporal behavior of the application. The way other real-time systems attempt to deal with asynchronous events is by responding immediately at the time of the event. The RTSJ takes a different approach, scheduling the response to asynchronous events to avoid disrupting the temporal integrity of the rest of the real-time application.

Asynchronous Transfer of Control

The RTSJ states:

> "Sometimes the real world changes so drastically (and asynchronously) that the current point of logic execution should be immediately and efficiently transferred to another location. The RTSJ includes a mechanism, which extends Java's exception handling to allow applications to programmatically change the locus of control of another Java thread. It is important to note that the RTSJ restricts this asynchronous transfer of control to logic specifically written with the assumption that its locus of control may asynchronously change."

The Asynchronous Transfer of Control (ATC) gives developers the ability to write new methods that can be interrupted.

Other real-time systems attempt to make provisions for asynchronous transfer of control, but do it in unsafe ways. The problem is that the event that triggers the ATC can occur when a thread has locked a resource, and if the thread terminates at that point, the resource will remain locked indefinitely, eventually leading to system malfunctions. Attempts to work around this problem by releasing all resources locked by the thread before killing it can lead to unpredictable behavior, too, because the resource might be released in an inconsistent state, resulting in incorrect functional behavior.

The safe way to handle ATC is to defer the asynchronous exception until the thread to be interrupted holds no resource locks. The RTSJ makes ATC safe by deferring exception handling during synchronized blocks, when the real-time system is actively managing tasks to prevent unbounded priority inversions.

The RTSJ does this by introducing the concept of a *deferred section*. Interrupting exceptions do not propagate immediately; they propagate only according to these deferred exceptions (that is, ATC is turned off as long as control is in a deferred section, which can be quite large).

There are two types of deferred sections:

- Synchronized blocks
- Methods that don't throw AsynchronouslyInterruptedException

That is to say, any section that is not a synchronized block can be made interruptible by including AsynchronouslyInterruptedException in its throws clause. Because they do not throw AsynchronouslyInterruptedException, all of the existing Java libraries are uninterruptible; they will all run all the way through to completion even if an AsyncEvent occurs, deferring execution of the AsyncEventHandler until the next *interruptible section* gets dispatched.

Physical Memory Access

The RTSJ states:

> "The PhysicalMemoryManager is available for use by the various physical memory accessor objects...to create objects of the correct type that are bound to areas of physical memory with the appropriate characteristics – or with appropriate accessor behavior. Examples of characteristics that might be specified are: DMA memory, accessors with byte swapping, etc.
>
> "The base implementation will provide a PhysicalMemoryManager and a set of PhysicalMemoryTypeFilter classes that correctly identify memory classes that are standard for the (OS, JVM, and processor) platform.
>
> "OEMs may provide PhysicalMemoryTypeFilter classes that allow additional characteristics of memory devices to be specified. Memory attributes that are configured may not be compatible with one another. For instance, copy-back cache enable may be incompatible with execute-only. In this case, the implementation of memory filters may detect conflicts and throw a MemoryTypeConflictException, but since filters are not part of the normative RTSJ, this exception is at best advisory."

Mackinac Performance

Although the RTSJ's definition of real-time excludes speed as a defining characteristic, performance is, of course, a consideration when choosing a real-time system to meet the needs of a particular real-time application. The next two subsections describe Mackinac performance in terms of two parameters: *throughput* and *latency and jitter*.

Throughput

For any real-time system, whether based on the Java platform or not, there is a cost associated with providing hard real-time capability – a trade-off between throughput and predictability. In the real-time developer community, the expectation is that a well-engineered real-time version of any software platform will suffer an average reduction of throughput in the range of 10%–20%.

The Mackinac real-time system is built on the Java HotSpot platform. Preliminary results across a small set of benchmarks show that the addition of real-time features in Mackinac reduces throughput by only 10% on average, while providing all of the benefits of the Java platform referred to earlier. Results are application-specific; the preliminary throughput results for Mackinac ranged from 40% slower to 2% faster than the Java HotSpot platform, depending on the application.

How does this performance compare to other, non-Java systems? The Java Hotspot platform (non-real-time) is faster than C++ (non-real-time), and the expectation is that both would incur roughly the same throughput penalty when converted to real-time systems, which means that the throughput of Mackinac should exceed that of a real-time implementation of C++ by about the same margin as in the non-real-time comparison.

Comparing the throughput of Mackinac to that of a non-object-oriented language, such as C, is more difficult because it is not clear exactly what constitutes a valid comparison. If the comparison is between the Hotspot platform and a high-performance C compiler, comparing a sequence of instructions that comes from an equivalent source code statement (that is, one statement from Java versus one statement from C), the Hotspot platform will be about the same as C in terms of straight-line throughput. Once again, the expectation is that both systems incur roughly the same throughput penalty when converted to real-time systems, which means that the throughput of Mackinac should be about the same as a real-time implementation of C, when the method of comparison is as described above.

This high level of throughput would, of course, be impossible if Mackinac interpreted Java byte code at run-time; instead, the code is compiled in advance. Mackinac does not use the just-in-time (JIT) compiler as it is normally used, however. Instead, at start-up, the compiler reads a file that contains a list of all the methods that have

to be pre-compiled. This is called *initialization-time compilation* (ITC). Non-real-time methods continue to be compiled on a JIT basis, but everything related to the real-time zones (HRT and SRT) uses ITC to avoid having the compiler introduce unpredictability into the system.

Latency and Jitter

The kind of canonical test used in the real-time space involves defining a periodic process that is released at constant intervals, as shown in Figure 14.

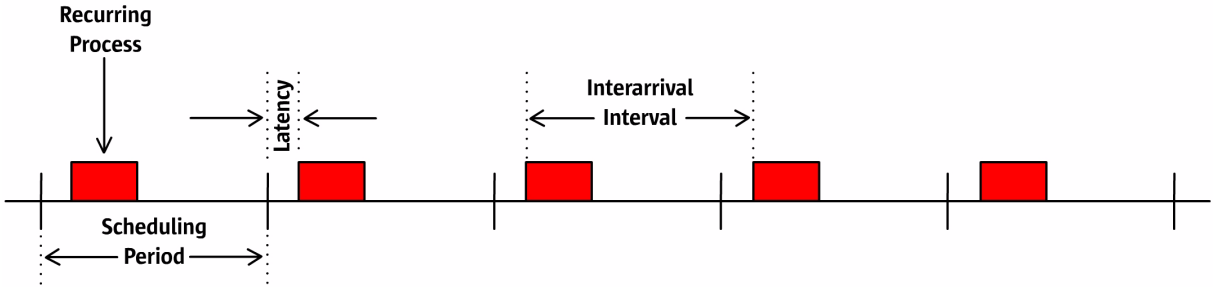


Figure 14. Timing the Real-Time Process

The tick marks represent the intervals at which each repetition of the process should begin, with the *scheduling period* being the distance between tick marks. *Latency* is the delay between the beginning of the scheduling period and the actual start of the process. As was explained earlier, the interarrival interval is the interval between the start of one recurring process and the start of the next one.

The variation in latency values over time will be distributed around some mean value, as shown in Figure 15.

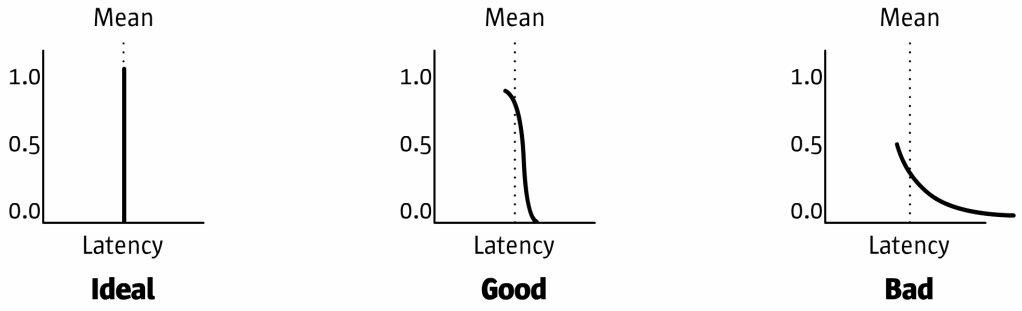


Figure 15. Variation in Latency

The magnitude of the median value is less important than the amount of variation in the distribution; that variation is called *jitter*. In the ideal case, there is no jitter—the variation in latency is zero. In the ideal case in which there is no jitter, the duration of the interarrival interval is always exactly equal to the duration of the scheduling period.

Reducing jitter is an important design consideration in any real-time system, not because latency is important in and of itself but because it is important to be able to predict with absolute certainty that a given process will always be complete before the end of the period in which it is executed. The more jitter there is, the more leeway must be allowed, which, in turn, degrades the efficiency of the system.

A critical section is defined as code in which only one thread at a time can be executing. The system cannot allow a thread in a critical section to be blocked. One thread has to leave the critical section before another can

enter it. When a thread leaves the critical section, if a high-priority task has work to be done, it will get processed before any lower-priority tasks that want to get into the critical section.

One problem facing any real-time system architect is the difficulty in determining what the maximum path length of the critical section is going to be. This maximum path length must include any device drivers. Figure 16 shows how the Mackinac project team chose to simplify the problem.

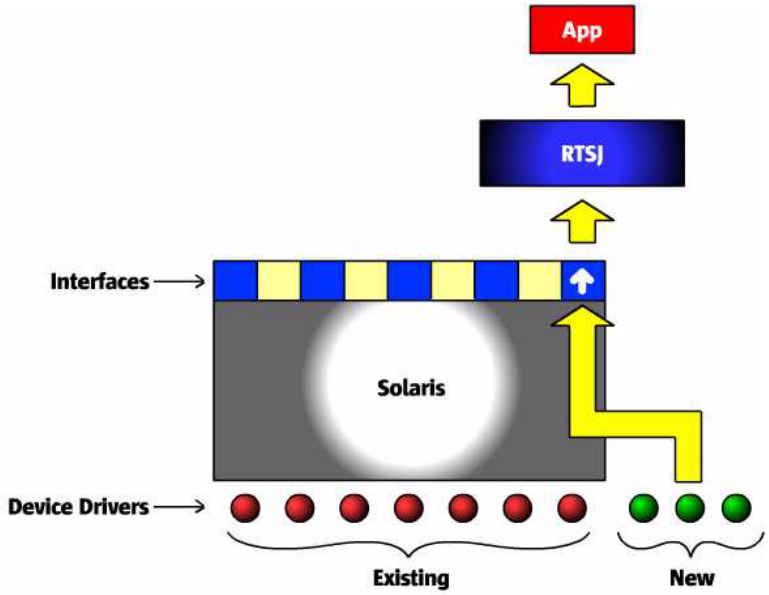


Figure 16. Routing Device Drivers to RTSJ

Project Mackinac was unique among real-time system implementers in having written three specialized device drivers and routed them through the Solaris IOCTL interface to let the new device drivers communicate directly with the real-time system. The alternative – and what all other implementers have done – is to analyze the whole kernel to see what the maximum path length is and how it affects the critical section. Enlarging the scope of the analysis in that way naturally introduces more variables, making the analysis less precise. The Mackinac approach has the dual advantage of shortening the critical section by bypassing unnecessary portions of the kernel, while also improving the precision of any feasibility analysis based on that critical section.

Conclusion

Trends in computing hardware and software, and in the market forces that influence computing behavior, are having a profound effect on the way we think about real-time software development.

- As real-time systems continue to grow in scale and complexity, old programming paradigms will not be able to keep pace with the needs of real-time developers. Meeting these emerging needs will require new paradigms, and new real-time platforms built on those paradigms.
- As real-time systems expand in scope, they will begin to integrate with existing business systems. Consequently, we will need to find ways to allow systems to share data, code, processes, memory, hardware, and so on. between real-time and non-real-time logic. These real-time and non-real-time components will need to be embraced in a comprehensive system architecture that allows them to interoperate seamlessly.
- As elements of real-time programming find their way into applications that, until now, have never required any real-time programming expertise, it will become increasingly difficult to view real-time developers and business application developers as members of two completely insular communities. Rather, ways will have to be found

to build bridges between these communities to facilitate the smooth flow of information from the smallest embedded device to the largest enterprise information system and vice versa.

Taken together, these trends will make it necessary for the real-time community to adopt real-time platforms that provide higher-level abstractions and advanced tools for functional and temporal programming. The Java platform is ideally suited to many of the needs facing the real-time community but, until now, has offered no support for real-time functionality. In writing the Real-time Specification for Java (RTSJ), the intent of the Java Community Process was to extend the benefits of Java to the real-time community, creating an architecture in which hard real-time (HRT), soft-real-time (SRT), and non-real-time (NRT) processes could coexist and share data.

Project Mackinac is the first commercial implementation of the RTSJ undertaken by Sun Microsystems. The initial release of Project Mackinac will address the needs of the midrange development project (desktop- or workstation-based). Future releases will include support for very small devices as well as support for enterprise-level real-time applications. Project Mackinac combines very competitive real-time performance with Sun's reputation for quality and long-term product support. Sun's reputation for designing and building robust, dependable, scalable architectures and systems is well deserved, and Sun's technological leadership in virtually every field of computer science is legendary.

Project Mackinac is a high-priority initiative managed and staffed entirely from within Sun Labs. With the resources of Sun Microsystems, the world leader in Java technology behind it, Project Mackinac intends to create a next-generation real-time platform that will continue to meet the needs of the real-time community well into the 21st century.

Sun Microsystems, Inc. 4150 Network Circle, Santa Clara, CA 95054 USA **Phone** 1-650-960-1300 or 1-800-555-9SUN **Web** sun.com

The Network is the Computer™

Sun Worldwide Sales Offices: Argentina +5411-4317-5600, Australia +61-2-9844-5000, Austria +43-1-60563-0, Belgium +32-2-704-8000, Brazil +55-11-5187-2100, Canada +905-477-6745, Chile +56-2-3724500, Colombia +571-629-2323 Commonwealth of Independent States +7-502-935-8411, Czech Republic +420-2-3300-9311, Denmark +45 4556 5000, Egypt +202-570-9442, Estonia +372-6-308-900, Finland +358-9-525-561, France +33-134-03-00-00, Germany +49-89-46008-0 Greece +30-1-618-8111, Hungary +36-1-489-8900, Iceland +354-563-3010, India–Bangalore +91-80-2298989/2295454; New Delhi +91-11-6106000; Mumbai +91-22-697-8111, Ireland +353-1-8055-666, Israel +972-9-9710500 Italy +39-02-641511, Japan +81-3-5717-5000, Kazakhstan +7-3272-466774, Korea +822-2193-5114, Latvia +371-750-3700, Lithuania +370-729-8468, Luxembourg +352-49 11 33 1, Malaysia +603-21161888, Mexico +52-5-258-6100 The Netherlands +00-31-33-45-15-000, New Zealand–Auckland +64-9-976-6800; Wellington +64-4-462-0780, Norway +47 23 36 96 00, People's Republic of China–Beijing +86-10-6803-5588; Chengdu +86-28-619-9333 Guangzhou +86-20-8755-5900; Shanghai +86-21-6466-1228; Hong Kong +852-2202-6688, Poland +48-22-8747800, Portugal +351-21-4134000, Russia +7-502-935-8411, Saudi Arabia +9661 273 4567, Singapore +65-6438-1888 Slovak Republic +421-2-4342-94-85, South Africa +27 11 256-6300, Spain +34-91-596-9900, Sweden +46-8-631-10-00, Switzerland–German 41-1-908-90-00; French 41-22-999-0444, Taiwan +886-2-8732-9933, Thailand +662-344-6888 Turkey +90-212-335-22-00, United Arab Emirates +9714-3366333, United Kingdom +44-1-276-20444, United States +1-800-555-9SUN or +1-650-960-1300, Venezuela +58-2-905-3800, or online at sun.com/store

Printed in USA 00/00 XX0000-0/#K