



High Integrity for Java™ Applications

The Safety Critical Solution for Java™ Developers

Technology Brief

CERTIFIABLE SAFETY CRITICAL SOLUTIONS FOR JAVA PROGRAMS

This technology brief provides an advance overview of technology which is currently under active development. Functionalities and target support are prioritized based on customer need.

In order to place your host / target needs high in the priority chain, or for specific delivery commitments, please contact your Aonix account manager.

Aonix, an industry leader in safety critical solutions for 20 years, is now making its RAVEN technology available for Java developers with the PERC Raven technology program. As the leaders of the Open Group Safety Critical Java Specification Initiative, Aonix is uniquely qualified to bring cutting-edge safety critical technology to our customers.

PERC Raven implements a small and very fast "bare-target" runtime system which offers the advantages of our PERC Pico technology but with additional safety-critical constraints and capabilities. PERC Raven is perfectly suited for hard real-time and safety-critical

From the start, the design and implementation of PERC Raven is focused on deterministic behavior, which is a key requirement for safety-critical systems. As a result, PERC Raven satisfies the highest levels of criticality, even Level A as defined in the DO-178B software safety guidelines required by the FAA for airborne systems.

Many Aonix customers have been using our safety-critical runtime environment for years (C-SMART and RAVEN). This runtime has been certified to Level A (as part of customers' applications) and can be found in avionic, railway, and nuclear power safety-critical applications already deployed and in development.



Whatever your real-time needs, Aonix is there to help with our world-class support team. We have the experience, tools and runtime environment for real-time Java that is second to none in the industry. PERC Raven has all the advantages of small size, fast speed, hard real-time response, safety-critical and high-reliability characteristics.

In addition to the tool chain itself, DO-178B certification artifacts can be created for the PERC Raven runtime and libraries, in support of end-application certification needs.

applications. It also provides the reliable, feature-rich toolset most critical system developers need to help them build an efficient, provable, verifiable and certifiable, deterministic real-time application.

Aonix continues to work closely with leaders in the Java safety critical community, including the US Navy, DARPA, and Expresso and HIJA initiatives.



Safety-Critical Development Guidelines for Real-Time Java

Safety-critical developers use a subset of the full hard real-time mission-critical capabilities.

Rule 1: Except Where Indicated to the Contrary, Use Hard Real-Time Programming Guidelines

In general, all of the hard real-time guidelines are appropriate for safety-critical development, except that certain practices acceptable for hard real-time mission-critical development should be avoided with safety critical software.

Rule 2: Use Only 28 Priority Levels for NoHeapRealtimeThread

The official RTSJ specification states that a compliant implementation must provide at least 28 priorities, but may support many more. For safety-critical development, application software should limit its use of priorities to the range from 1 though 28. Vendors can readily support this priority range as a standard safety-critical platform.

Rule 3: Use Only Instances of COMPLIANT-mode NoHeapRealtimeThread

In safety-critical systems, the sharing of data and control between native code and safety-critical Java code is strongly discouraged.

Rule 4: Prohibit Use of @OmitSubscriptChecking Annotation

In safety-critical code, turning off subscript checking is strongly discouraged, even though static analysis of the program presumably has proven that the program will not attempt to access invalid array elements. In safety-critical systems, the key benefit of subscript checking is to prevent an error in one component from propagating to other components.

Rule 5: Prohibit Use of @OmitScopeChecking Annotation

In safety-critical code, turning off assignment scope checking in methods that permit @AllowChecked-ScopedLinks is strongly discouraged, even though careful inspection of the program may have determined that the program will not attempt to make assignments that would violate nested scoping rules.

Rule 6: Prohibit Invocation of Methods Declared with @AllowCheckedScopedLinks Annotation

This annotation is designed to allow programmers to use practices that cannot be certified safe by automatic static theorem provers. Thus, there is a risk that any software making use of this annotation will abort with a run-time exception. Allow this practice only in safety-critical systems for which developers are able to provide absolute proof that run-time exceptions will not be thrown.

Rule 7: Require All Code To Be @StaticAnalyzable

In hard real-time mission-critical code, the use of the @StaticAnalyzable annotation is entirely optional. In safety-critical code, we require all components to have this annotation, and for all relevant modes of analysis to have a true value for the enforce_time_analysis, enforce_memory_analysis, and enforce_non_blocking attributes.

Rule 8: Require All Classes with Synchronized Methods to Inherit PCP or Atomic

The safety-critical profile does not allow the use of priority inheritance locking.

Rule 9: Prohibit Dynamic Class Loading

While dynamic class loading may be supported in the hard real-time mission-critical domain, it should be strictly avoided in safety-critical software.

Rule 10: Prohibit Use of Blocking Libraries

Because of difficulties analyzing blocking interaction times when software components contend for shared resources, all services that might block are forbidden in safety-critical code.

Rule 11: Prohibit Use of PriorityInheritance MonitorControl policy

Priority inheritance is more difficult to certify, more complicated to implement, and less efficient than priority ceiling emulation. Also, the implementation of priority inheritance introduces synchronization overhead delays that are proportional to the complexity of the application, rather than a function only of the system configuration. Analyzing these delays is particularly difficult. For all of these reasons, we prohibit its use in safety-critical software systems.

Rule 12: Do Not Share Safety-Critical Objects With a Traditional Java Virtual Machine

Combining safety-critical code with traditional Java code using the @TraditionalJavaMethod and @TraditionalJavaShared conventions compromises the integrity of the safety-certification artifacts. This practice is therefore strictly forbidden.

Rule 13: Use Development Tools to Enforce Consistency With Safety-Critical Guidelines

To enforce that programmers make proper use of the safety-critical subset and that all code is consistent with the intent of the hard real-time programming annotations described in this section, use special bytecode verification tools that help assure reliable and efficient implementation of programmer intent

(Reprinted from Draft Guidelines for Scalable Java Development of Real-Time Systems, May 6, 2005) © 2005 Aonix.

051706

To obtain more information, please contact Aonix at www.aonix.com or your local Aonix office.



NORTH AMERICA
Phone: 800.97-AONIX
Fax: 858.824.0212
Email: info@aonix.com

FRANCE
Phone: +33 (0) 1 4148-1000
Fax: +33 (0) 1 4148-1020
Email: info@aonix.fr

UNITED KINGDOM
Phone: +44 (0) 1491 415000
Fax: +44 (0) 1491 571866
Email: info@aonix.co.uk

GERMANY
Phone: +49 (0) 7243 5318-0
Fax: +49 (0) 7243 5318-78
Email: info@aonix.de