Safety-Critical Java Technology Specification

JSR-302

Version 0.109
27 January 2017
Draft

Expert Group Membership

Each Expert Group member is listed with the organization represented, if any.

Core Group

Doug Locke (LC Systems Services Inc., representing The Open Group - Specification Lead)
B. Scott Andersen (Self - employed by Verocel)
Ben Brosgol (Self - employed by AdaCore)
Mike Fulton (IBM)
Thomas Henties (Siemens AG)
James J. Hunt (aicas GmbH)
Johan Olmütz Nielsen (DDC-I, Inc.)
Kelvin Nilsen (Atego)
Anders Ravn (Self - employed by Aalborg University)
Martin Schoeberl (Self - employed by T.U. Copenhagen)
Jan Vitek (Self - employed by Purdue U.)
Andy Wellings (Self - employed by U. of York)

Consulting Group

Robert Allen (Boeing)
Greg Bollella (Oracle)
Arthur Cook (Self - employed by Alion Science & Technology)
Allen Goldberg (Self - employed by UC Santa Cruz)
David Hardin (Rockwell Collins, Inc.)
Joyce Tokar (Self - employed by Pyrrhusoft)

Contents

Document Control

Version	Status	Date
0.1	Draft	Uncontrolled draft
0.2	Draft	Uncontrolled draft
0.3	Draft	Uncontrolled draft
0.4	Draft	25 July 2008
0.5	Draft	Work-in-progress
0.6	Draft	Work-in-progress
0.65	Draft	San Diego Feb 2009
0.66	Draft	London May 2009
0.67	Draft	Pre-Toronto July 2009
0.68	Draft	Toronto July 2009
0.69	Draft	Pre-Madrid Oct 2009
0.73	Draft	Pre-Karlsruhe Apr 2010
0.75	Draft	Karlsruhe May 2010
0.77	Draft	Boston July 2010
0.78	First Release	JCP October 2010
0.79	Draft	May 2011
0.80	Draft	November 2011
0.94	Second Release	25 June 2013
0.95	Draft	26 January 2014
0.96	Draft	9 October 2014
0.100	Draft	27 December 2014
0.101	Draft	21 April 2015
0.105	Draft	14 December 2015
0.106	Draft	10 June 2016
0.107	Draft	24 September 2016
0.108	Draft	6 January 2017
0.109	Third Early Draft Review	27 January 2017

Executive Summary

This Safety-Critical Java Specification (JSR-302), based on the Real-Time Specification for Java (JSR-1), defines a set of Java services that are designed to be usable by applications requiring some level of safety certification. The specification is targeted to a wide variety of very demanding certification paradigms such as the safety-critical requirements of DO-178C, Level A.

This specification presents a set of Java classes providing for safety-critical application start up, concurrency, scheduling, synchronization, input/output, memory management, timer management, interrupt processing, native interfaces, and exceptions. To enhance the certifiability of applications constructed to conform to this specification, this specification also presents a set of annotations that can be used to permit static checking for applications to guarantee that the application exhibits certain safety properties.

To enhance the portability of safety-critical applications across different implementations of this specification, this specification also lists a minimal set of Java libraries that must be provided by conforming implementations.

Chapter 1

Introduction

Safety-Critical Java (SCJ) technology, based on the Real-Time Specification for Java (RTSJ) [2] has been designed to address the general needs of adapting Java technology for use in safety-critical applications. As Java has matured, it has become increasingly desirable to leverage Java technology within applications that require not only predictable performance and behavior, but also high reliability. When such performance and reliability are required to protect property and human life, such systems are said to be safety-critical. This document specifies a Java technology appropriate for safety-critical systems called Safety-Critical Java (SCJ).

Safety-critical systems can be defined as systems in which an incorrect response or an incorrectly timed response can result in significant loss to its users; in the most extreme case, loss of life may result from such failures. For this reason, safety-critical applications require an exceedingly rigorous validation and certification process. Such certification processes are often required by legal statute or by certification authorities. For example, in the United States, the Federal Aviation Administration requires that safety-critical software be certified using the Software Considerations in Airborne Systems and Equipment Certification (DO-178C [6] or in Europe, the ED-12C [7]) standard controlled by an independent organization.

The development of certification evidence for a software work-product used within a safety-critical software system is extremely time-consuming and expensive. Most safety-critical software development projects are carefully designed to reduce the application size and scope to its most minimal form to help manage the costs associated with the development of certification evidence. Examples of the resulting restrictions may include the elimination or severe limitations on recursion and the rigorous use of memory, especially heap space, to ensure that out-of-memory conditions are precluded.

In the context of Java technology, as compared to other Java application paradigms, this requires a smaller and highly predictable set of Java virtual machines and li-

braries. They must be smaller and highly predictable both to enhance their certifiability and to permit meeting tight safety-critical application performance requirements when running with Java run-time environments and libraries. Additionally, safety-critical applications must exhibit freedom from any exceptions that cannot be successfully handled. This requires, for example, that there be no memory access errors at run-time.

This safety-critical specification is designed to enable the creation of safety-critical applications, built using safety-critical Java infrastructures, and using safety-critical libraries, amenable to certification under DO-178C, Level A, as well as other safety-critical standards.

1.1 Definitions, Background, and Scope

The field of safety-critical software development makes use of a number of specialized terms. Though definitions for these terms may vary throughout safety-critical systems literature, there are some concepts key to this discussion that can be crisply defined. Below is a set of specific terms and the associated definitions that provide important background information for understanding this standard:

Storey [8] provides several useful definitions:

- *Safety* is a property of a system that a failure in the operation of the system will not endanger human life or its environment.
- The term *safety-critical system* refers to a system of high criticality (e.g. in DEF STAN 00-55[9] it relates to Safety Integrity Level 4) in which the safety of the related equipment and its environment is assured. A safety-critical system is generally one which carries an extremely high level of assurance of its safety.
- *Safety integrity* refers to the likelihood of a safety-critical system satisfactorily performing its required safety functions under all stated conditions within a stated period of time.

Some additional definitions from Burns and Wellings [1] are useful as well:

- *Hard real-time components* are those where it is imperative that output responses to input stimuli occur within a specified deadline.
- *Soft real-time components* are those where meeting output response time requirements is important, but where the system will still function correctly if the responses are occasionally late.

In many safety-critical contexts, multiple levels of safety-critical certification are defined. For example, in the aviation industry, the DO-178C and ED-12C standards define the following software levels

- *Level A:* Software whose anomalous behavior would cause or contribute to a failure of system function resulting in a catastrophic failure condition for the aircraft. A catastrophic failure is one which would prevent continued safe flight and landing.
- *Level B:* Software whose anomalous behavior would cause or contribute to a failure of system function resulting in a hazardous/severe major failure condition for the aircraft. A hazardous/severe major failure is one which would reduce the capability of the aircraft or the ability of the crew to cope with adverse operating conditions to the extent that there would be, for example, a large reduction in safety margins or potentially fatal injuries to a small number of the aircrafts' occupants.
- *Level C:* Software whose anomalous behavior would cause or contribute to a failure of system function resulting in a major failure condition for the aircraft. A major failure is one which would reduce the capability of the aircraft or the ability of the crew to cope with adverse operating conditions to the extent that there would be, for example, a significant reduction in safety margins or discomfort to occupants, possibly including injuries.
- *Level D:* Software whose anomalous behavior would cause or contribute to a failure of system function resulting in a minor failure condition for the aircraft. A minor failure is one which would not significantly reduce aircraft safety or functionality.
- *Level E:* Software whose anomalous behavior would cause or contribute to a failure of system function with no effect on aircraft operational capability.

Note that Level D and Level E systems have been successfully constructed using standard Java technology without the aid of this specification. This specification is oriented toward the higher levels of certification, although this standard does not, by itself, assure that a conforming application will meet any level of certification.

Other standards have similarly defined levels and also add a probability of such a failure occurring. For example, in IEC 61508 [4], the maximum probability of a catastrophic failure (for Level A) is defined to be between 10^{-5} and 10^{-4} per year per system. In DEF STANDARD 00-56 [10], Safety Integrity Levels (SILs) are defined in terms of the predicted frequency of failures and the resulting severity of any resulting accident (see Figure 1).

The type of verification techniques that must be used to show that a software component meets its specification will depend on the SIL that has been assigned to that component. For example, Level A and B software might be constrained so it can be subjected to various static analysis techniques (such as control flow analysis).

Evidence may also be demanded for structural coverage analysis, an analysis of the execution flows for the software that determines that all paths through the software have been tested or analyzed, and that there is an absence of unintended function

Failure Probability	Accident Severity			
	Catastrophic	Critical	Marginal	Negligible
Frequent	4	4	3	2
Probable	4	3	3	2
Occasional	3	3	2	2
Remote	3	2	2	1
Improbable	2	2	1	1

Figure 1.1: DEF STANDARD 00-56 Safety Integrity Levels

within the code. Additionally, decisions affecting control flow may also need to be examined and evidence produced to show that all decisions, and perhaps even conditions within those decisions, have been exercised though testing or analysis. Specific techniques such as Modified Condition Decision Coverage (MCDC) [6] may be mandated as part of this analysis.

The type and level of structural coverage analysis (within a requirements-based testing framework) might be different for different certification levels. For example in DO-178C MCDC is compulsory at Level A but optional at Level B; only statement level coverage is required at Level C. Also, whether or not the analysis and testing must be performed independently (a requirement that the developer of an artifact must not also be its reviewer) may vary among levels.

It is important to understand that this specification can not, and will not attempt to ensure that a conforming application or implementation will meet the demands of certification under any safety-critical standard, including DO-178C. Rather, this specification is intended to enable a conforming application and implementation to be certifiable when all conditions defined by a safety-critical standard (such as DO-178C) are also met. It is the responsibility of the developer to understand and fulfill the specific requirements of the applicable standards. By implication, it remains the responsibility of application and implementation developers to create the "certification artifacts", i.e., the required documentation for a certification authority that will be needed to complete the application's safety certification.

The requirements imposed by safety-critical standards such as DO-178C have been used to identify the capabilities and limitations likely needed by a safety-critical application developer using Java technology. Additionally, the objectives identified within DO-178C for Level A software have been used to guide key decisions within

this Safety-Critical Java framework because Level A represents one of the most stringent standards in use today. Systems amenable to certification under DO-178C Level A are also likely to able to attain certification under similar safety standards.

The use of five levels in the DO-178C reflects the fact that the safety requirements of any system, including its software, occupy a place on a wide spectrum of safety properties. At one end of this spectrum are systems whose failure could potentially cause the loss of human life, such as those covered by DO-178C, Level A. At the other end of the spectrum are systems with no safety responsibilities, such as an in-flight entertainment system.

The next major position on this spectrum below safety-critical is that of mission-critical software. Mission-critical software consists of software whose failure would result in the loss of key capabilities needed to successfully carry out the purpose of the software such that a failure could cause considerable financial loss, loss of prestige, or loss of some other value. An example of a mission-critical system would be a Mars rover.

Unfortunately, there is no fully accepted definition of mission-critical real-time software, although there is broad agreement that mission-critical software is deemed vital for the success of the enterprise using that software, and any failure will have a significant negative impact on the enterprise (possibly even its continuing existence). Safety-critical software is clearly also mission-critical software in the sense that failure of safety-critical software is likely to result in a mission failure. In general however, mission-critical software may not (directly) cause loss of life and therefore will probably not be subject to as rigorous a development and assessment/certification process as safety-critical software. The authors of this specification have considerable interest in mission-critical systems, and consider it likely that a similar (but broader) specification may be created addressing mission-critical systems, but a Java specification for mission-critical systems is explicitly beyond the scope of this effort.

1.2 Additional Constraints on Java Technology

There are many issues associated with the use of Java technology in a safety-critical system but the two largest issues are related to the management of memory and concurrency. This specification addresses both of these architectural issues and defines a model based on the one described in the RTSJ. Six major additional constraints are imposed on the RTSJ model as described below.

1. The safety-critical software community is conservative in adopting new technologies, approaches, and architectures. This safety-critical Java software specification is constrained to respect both the traditions of the Java technology

community and the safety-critical systems community. The Ada Ravenscar profile is an example of a language and technology that has been constrained to meet the needs of the safety-critical software community, but it was accepted only after the definition was stringently defined and simplified from its pure Ada roots, especially in regards to the models of concurrency that were provided. Constraints on the usage of dynamic memory allocation, and especially reallocation, are also imposed to mitigate out-of-memory conditions and simplify analysis of memory usage during development of certification evidence. Severe constraints on concurrency and heap usage, not typical of traditional Java technology-based applications, are commonplace within the safety-critical software community.

2. The safety-critical Java technology memory management and concurrency specified here is based on the technology within the RTSJ (version 2.0) and Java technology version 8.0. With very minor exceptions delineated later in this specification (See Chapter 2), a safety-critical Java application constructed in accordance to this specification will execute correctly (although not with the same performance) on an RTSJ compliant platform when the Safety-Critical Java libraries specified herein are provided.

3. New classes are defined in this specification, but these classes are designed to be implementable using the facilities of the RTSJ. New classes are generally introduced when the use of the native RTSJ facilities would obfuscate or add complexity to a conforming application or implementation, or when it is necessary to increase the safety of an interface.

4. Annotations have been defined to provide a means of documenting a few of the critical memory management and concurrency assumptions made by the application programmer to facilitate off-line tools in identifying certain errors prior to run-time.

5. Some widely used Java capabilities are omitted from this specification to enable the certifiability of conforming applications and implementations. Dynamic class loading is not required. Finalizers will not be executed. Many Java and RTSJ classes and methods are omitted. The procedure for starting an application differs from other Java platforms such as that defined in the RTSJ. Unlike the RTSJ, synchronization is required to support priority ceiling emulation, and a conforming implementation need not support priority inheritance. Further, the RTSJ requires that a ThrowBoundaryError exception be created in a parent scoped memory if an exception is thrown but unhandled while executing in a child scoped memory. This specification defines the same behavior except that the ThrowBoundaryError exception behaves as if it had been preallocated on a per-schedulable object basis.

6. This specification takes no position on whether a safety-critical conforming application is interpreted or is compiled to object code and executed using a run-time environment.

1.3 Key Specification Terms

A number of specific terms are used in this specification to identify the mandatory behavior of compliant implementations and applications, and also to identify behavior which is not mandatory for implementations and applications. These terms are:

1. *Implementation Defined* — When the phrase "implementation defined" is used in this specification, it means that the antecedent can be designed and implemented in any way that the implementation's designers wish, but that the details of its functionality must be documented and made available to users and prospective users of the implementation.

2. *Unspecified* — When the phrase "unspecified" is used in this specification, it means that the antecedent can be designed and implemented in any way that the implementation's designers wish, and that the application must tolerate any behavior.

3. *Shall* — When the word "shall" is used in this specification, it means that a requirement is stated that is mandatory for the implementation or the application as determined by the context.

4. *May* — When the word "may" is used in this specification, it means that a preference or possible action is stated, but that it is not mandatory for the implementation or the application as determined by the context.

5. *Implementation* — An SCJ implementation is a vendor-supplied infrastructure providing all the tools needed for developing and executing a safety-critical application program. For example, a safety-critical implementation would include a Java virtual machine or run-time environment and analysis tools for use by a safety-critical application.

6. *Application* — An SCJ application is a specific safety-critical program to perform a safety-critical task. For example, a flight control system is a safety-critical application.

1.4 Specification Context

This specification defines the requirements for SCJ conformant applications and implementations and is accompanied by two other components: a Reference Implementaion (RI) and a Technology Compatibility Kit (TCK).

The RI is an actual implementation of the mandatory interfaces of this specification that satisfies these requirements and thus permits users and implementers of this specification to fully understand the specification in the context of an executing program, as well as providing a platform for experimenting with application designs that conform to this specfication. The RI is available under an open source license.

The TCK consists of Java application code that conforms with this specification and serves to test whether an implementation is conformant to this specification. Conforming implementations must correctly execute the entire TCK in order to claim SCJ conformance. The TCK source code for SCJ is publicly available under an open source license, but it must be understood that an implementation must correctly execute the official TCK with no changes in order to claim SCJ conformance.

Conforming implementations of this specification must not only provide the Java infrastructure needed to provide the SCJ classes and methods of this specification to conforming SCJ applications, but they must also provide a *Checker* utility to check that the application's annotations correctly define the application's associated safety properties.

The specification contains "normative" and "non-normative" content. Normative content defines the syntax and semantics of an SCJ compliant implementation or application. Non-normative content is provided only for clarity or to assist in understanding the normative content of the specification. Chapters 1 and 2 are non-normative. For each of the remaining chapters of this specification, the chapter's Introduction section states which sections of that chapter are normative.

1.5 Overview of the Remainder of the Document

This specification is focused on defining the constraints on the Java technology necessary to facilitate the development of safety-critical applications. The organization of this document is:

Chapter 2 presents the programming model and introduces the concept of a mission, and the three compliance levels, Level 0, Level 1, and Level 2. These compliance levels provide application developers with varying levels of sophistication in the programming environment with Level 0 being the most simple (and limiting), and Level 2 offering the greatest number of facilities.

Chapter 3 presents the mission life cycle and describes how a mission (an application or portion of an application) is initialized, run, and terminated. This chapter also describes how to sequence several missions, and how an application can create and execute multiple missions under some circumstances.

Chapter 4 presents the concurrency and scheduling models including the types and handling of events (periodic and aperiodic). Threads and schedulable objects are also discussed, as well as multiprocessors.

Chapter 5 presents the external event handling model, including interrupts, and their relationships.

Chapter 6 presents SCJ support for simple, low-complexity I/O.

Chapter 7 presents memory management, and specifically how memory handling differs from that in the RTSJ. Control mechanisms for memory area scope and lifetimes are identified.

Chapter 8 presents clocks, timers, and time.

Chapter 9 presents the Java metadata annotation system and its use within the SCJ class library.

Chapter 10 presents Java Native Interface (JNI) usage within SCJ applications.

Chapter 11 presents exceptions and the exception model for SCJ applications.

Chapter 12 presents class libraries for SCJ applications.

The required interfaces from standard Java, the RTSJ, and the SCJ library classes are included in the Appendix.

Chapter 2

Programming Model

This Safety-Critical Java (SCJ) specification, in contrast to the Real-Time Specification for Java (RTSJ), imposes significant limitations on how a developer structures an application, and supports only a few relatively simple software models in terms of concurrency, synchronization, memory, etc. This is appropriate because safety-critical applications must generally conform to rigorous certification requirements, so therefore they generally use much simpler programming models that are amenable to certification than that permitted under standard Java technology or the RTSJ.

This specification is based on the Java 8.0 language reference and the RTSJ. Specifically, this specification can be considered to define a subset of the Java environment and the RTSJ (version 2.0) to support safety-critical systems. It is intended that SCJ-compliant applications should be readily portable from an SCJ environment to a RTSJ environment.

In this specification, a flexible programming model is defined that is intended to be sufficiently limited to enable certification under such standards as DO-178C Level A. This is accomplished by defining concepts such as a mission, limited start up procedures, and specific levels of compliance. In addition, a set of special annotations is described that are intended for use by vendor-supplied and/or third-party tools to perform static off-line analysis that can ensure certain correctness properties for safety-critical applications.

Because safety-critical systems are typically also hard real-time systems (i.e., they have time constraints and deadlines that must be met predictably), methods implemented according to this specification should have predictably bounded execution behavior. Worst case execution time and other bounding behavior is dependent on the application and its SCJ execution environment.

2.1 The Mission Concept

Under this specification, a compliant application will consist of one or more *missions*. A mission consists of a set of schedulable objects. A schedulable object consists of a sequence of code that is scheduled by a fixed-priority scheduler (or a cyclic executive in a Level 0 SCJ implementation) included with the SCJ implementation. Schedulable objects in this specification are derived from the schedulable objects defined by the RTSJ.

For each mission, a specific block of memory is defined called *mission memory*. Objects created in mission memory persist, and their resources will not be reclaimed, until the mission is terminated, If the application chooses to terminate a mission, this specification provides for the application to select another mission to be executed, erasing the current mission's mission memory. If the application does not provide a sequence of missions, it can either avoid terminating the mission, or stop all processing.

Conforming implementations are not required to support dynamic class loading. Classses visible within a mission are unexceptionally referenceable. Class initialization must be completed before any part of any mission runs, including its initialization phase (described below). There is no requirement that classes, once loaded, must ever be removed, nor that their resources be reclaimed. A properly formed SCJ program should not have cyclic dependencies within class initialization code. For further details on the requirements for an SCJ application, see Section 3.2.1.

Each mission starts in an *initialization phase* during which objects may be allocated in mission memory and immortal memory by an application. Immortal memory is never reclaimed at all, while Mission memory is reclaimed at the termination of a mission and before the start of the next mission. When a mission's initialization has completed, its *execution phase* is entered. During the execution phase an application may access objects in mission memory and immortal memory, but will usually not create new objects in mission memory or immortal memory. If the application subsequently terminates a mission, a *clean up phase* is entered. During the clean up phase, each schedulable object runs to completion or an explicit waiting point, and thereafter an application-defined set of clean up methods is executed. Objects in Immortal memory are not affected by sequencing to the next mission (if one exists), but objects in Mission memory will be removed before the next mission is started.

When one of a mission's schedulable objects is started, its initial memory area is a private memory area that is entered when the schedulable object is *released*, and is exited (i.e., emptied) before the schedulable object is released again. (See Section 7.2.1 for details) This private memory area is not shared with other schedulable objects.

2.2 Compliance Levels

Safety-critical software application complexity varies greatly. At one end of this range, many safety-critical applications contain only a single thread, support only a single function, and may have only simple timing constraints. At the other end of this range, highly complex applications have multiple modes of operation, may contain multiple (nested) missions, and must satisfy complex timing constraints. While a single safety-critical Java implementation supporting this entire range could be constructed, it would likely be overly expensive and resource intensive.

Minimizing complexity is especially important in safety-critical applications because both the application and the infrastructure, including the Java runtime environment, must undergo certification. The cost of certification of both the application and the infrastructure is highly sensitive to their complexity, so enabling the construction of simpler applications and infrastructures is highly desirable.

For these reasons, this specification defines three compliance levels to which both implementations and applications may conform. This specification refers to them as Level 0, Level 1, and Level 2, where Level 0 supports the simplest applications and Level 2 supports more complex ones. The cost and difficulty of achieving any given certification level is expected to be higher at Level 2 than at Level 1 or Level 0.

These three compliance levels have no relationship with the safety levels defined by standards such as DO-178C.

The requirements for each Level are designed to ensure that properly synchronized SCJ missions at any Level will execute correctly on any compliant implementation that is capable of supporting that Level or a higher Level. Thus, for example, a Level 1 application must be able to run correctly on an implementation supporting either Level 1 or Level 2. Conversely, implementations at higher levels must be able to correctly execute applications requiring support at that level or below. It must be noted that while Level 0 applications execute under a cyclic executive structure in a Level 0 implementation, a Level 0 application executing in a Level 1 or Level 2 implementation will not be executing under a cyclic executive. See Chapter 4 for a detailed discussion of SCJ execution models.

At each Level, an application consists of a sequence of Missions. If a sequence consists of more than one Mission, the next Mission is determined and run by an application-defined mission sequencer.

The definition of each level includes the types of schedulable objects (e.g., PeriodicEventHandler, AperiodicEventHandler) permitted at that level, the types of synchronization that can be used, and other permitted capabilities.

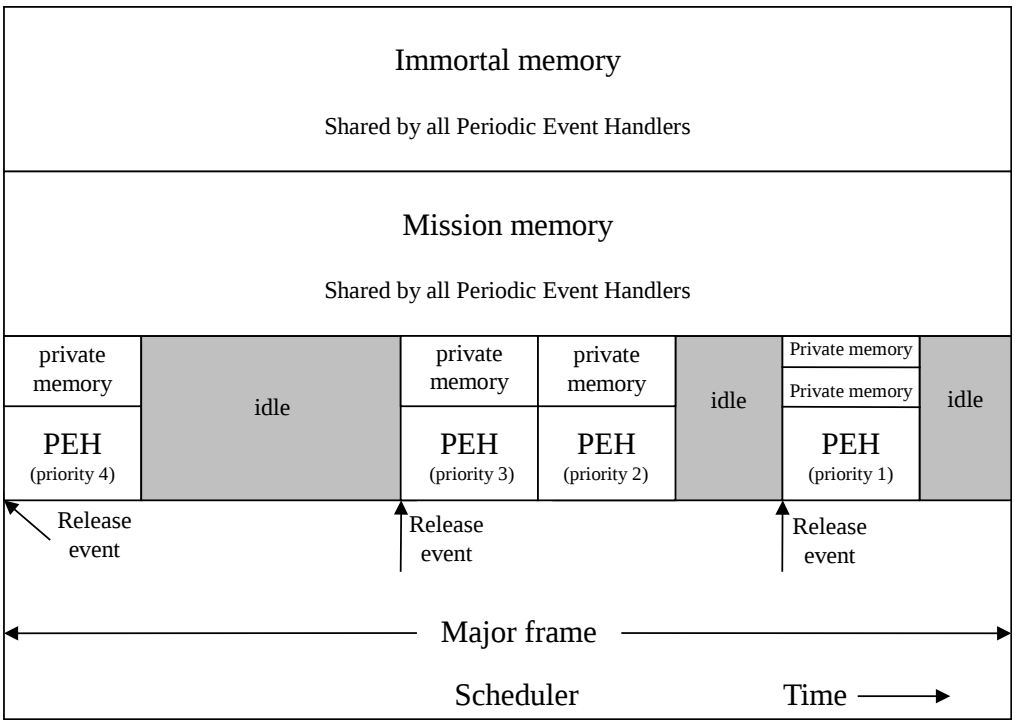

Figure 2.1: Level 0 [Cyclic Executive]

2.2.1 Level 0

A Level 0 application's programming model is a familiar model often described as a timeline model, a frame-based model, or a cyclic executive model. In this model, a mission can be thought of as a set of computations, each of which is executed periodically in a precise, clock-driven timeline, and processed repetitively throughout the mission. This model assumes that the application is designed to ensure that the execution of each frame is completed within that frame.

Figure 2.1 illustrates the execution of a simple Level 0 application, including its memory allocation. It shows four periodic event handlers being released, each with its own private memory that is erased after each release. Each periodic event handler release is triggered by a timer under the control of a cyclic executive. The entire schedule is repeated at a fixed period (i.e., a major cycle). The timer values and major cycle period are defined in a schedule provided by the application. The priorities shown are disregarded when running under a Level 0 implementation because the cyclic schedule is specifically defined, but would be used if the Level 0 application were run under a Level 1 or Level 2 implementation. The figure shows only a single mission, but it is also possible to run multiple missions sequentially.

A Level 0 application's schedulable objects consist only of PeriodicEventHandler (PEH)objects that are derived from the ManagedSchedulable (See Chapter 4.5.6 for details) class. Although they are not used if the Level 0 application is executed by a Level 0 implementation, each PEH should have a period, priority, and start time relative to its period. A schedule of all PEHs is constructed by either the application designer or by an offline tool provided with the implementation.

Thus, in a Level 0 implementation, all PEHs execute sequentially as if they were all executing within a single infrastructure thread. This enforces the sequentiality of every PEH, so the implementation can safely ignore synchronization. The application developer, however, is strongly encouraged to include the synchronization required to safely support its shared objects so the application would maintains consistency regardless of whether it is running on a Level 0, Level 1, or a Level 2 implementation.

The use of a single infrastructure thread to run all PEHs without synchronization implies that a Level 0 application runs only on a single CPU. If more than one CPU is present, it is necessary that the state managed by a Level 0 application not be shared by any application running on another CPU. This specification describes the semantics for a single application; interactions, if any, among multiple applications running concurrently in a system are beyond the scope of this specification.

The methods Object.wait and Object.notify are not available to a Level 0 application. Applications should also avoid blocking because all of the application's PEHs are executing in turn as if they were running in a single thread.

Each PEH has a private scoped memory area, an instance of private memory, created for it before its first release, that will be entered and exited each time it is released. A Level 0 application can create private memory areas directly nested within the provided private memory area. It can enter and exit them, but it may not share them with any other PEH.

2.2.2 Level 1

A Level 1 application uses a familiar multitasking programming model consisting of a single mission sequence. Each mission contains a set of concurrent computations, each with a priority, running under control of a fixed-priority preemptive scheduler. The computations are performed by a set of ManagedSchedulable objects consisting only of PEHs and/or AperiodicEventHandler instances (APEHs). An application shares objects in mission memory and immortal memory among its PEHs and APEHs, using synchronized methods to maintain the integrity of these objects. The methods Object.wait and Object.notify are not available.

Each PEH or APEH has a private scoped memory area, an instance of private memory, created for it before its first release that will be entered and exited at each release.

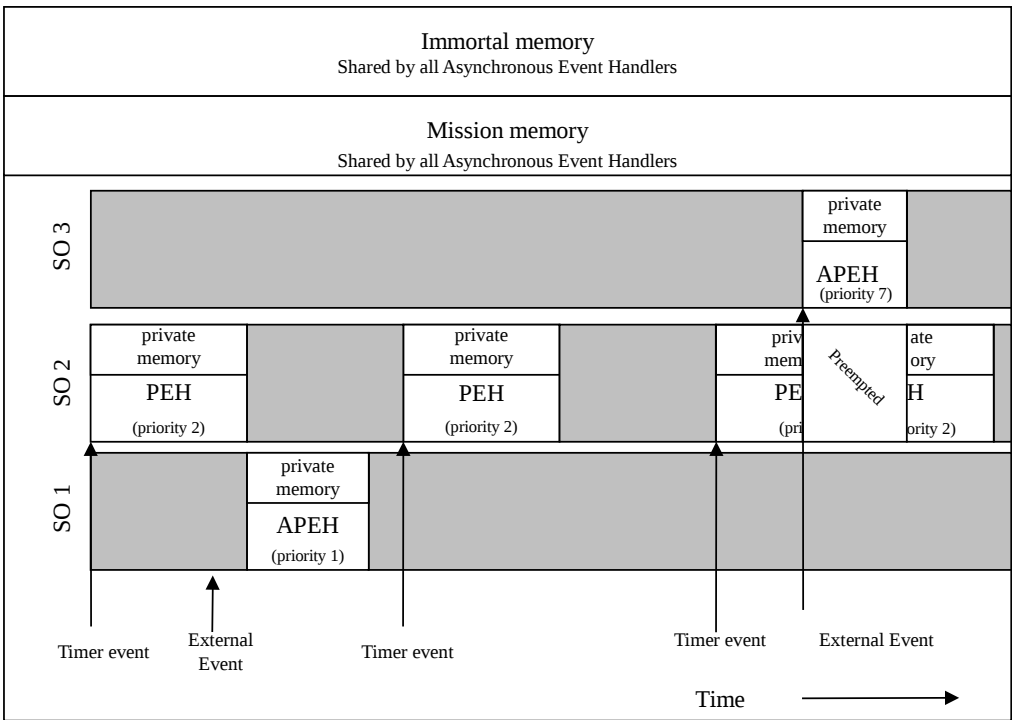

Figure 2.2: Level 1 [Single Mission]

During execution, the PEH or APEH may create, enter, and/or exit one or more other private memory areas, but these memory areas may not be shared among them.

Figure 2.2 illustrates the execution of a simple application running on a single processor with a single mission, including its memory allocation. It shows three schedulable objects, SO1, SO2, and SO3, each with a priority and a private memory area that is emptied before each release. The fixed priority preemptible scheduler executes them in priority order. When a higher priority schedulable object becomes ready to run, it may preempt a lower priority object at any time as shown when SO3 at priority 7 preempts SO2 at priority 2.

2.2.3 Level 2

A Level 2 application starts with a single mission, but may create and execute additional missions concurrently with the initial mission. Computations in Level 2 missions are performed by a set of ManagedSchedulable objects consisting of PEHs, APEHs, and/or managed threads which are similar to RTSJ no-heap real-time threads. Each child mission has its own mission memory.

Each Level 2 ManagedSchedulable object has a private scoped memory area created

before its first release. For PEHs and APEHs, the private scoped memory area will be entered and exited at each release; this memory area will be cleared before its next release. For no-heap real-time threads, the private scoped memory area will be entered when it starts its run method and exited when the run method terminates. During execution, each ManagedSchedulable object may create, enter, and/or exit one or more other scoped memory areas, but these scoped memory areas may not be shared among its schedulable objects. A Level 2 application may use Object.wait and Object.notify.

Figure 2.3 illustrates the execution of a simple application with one nested mission, including its memory allocation. Two missions are shown. Mission 1 starts first and contains three ManagedSchedulable objects. Mission 2 starts later and contains two schedulable objects. The priorities of each object determine the order of execution, regardless of which mission contains each object. For example, a preemption situation is shown in which SO2 in Mission 1 becomes ready to run and preempts SO1 in Mission 2. Note that a Level 2 application is permitted to use a ManagedThread object.

2.3 SCJ Annotations

To enable a level of static analyzability for safety-critical applications using this specification, a number of annotations following the rules of Java Metadata Annotations are defined and used throughout this specification. A complete description of these annotations is provided in Chapter 9.

One pervasive annotation in this specification is @SCJAllowed(level). It marks the minimum Level at which any specific class, interface, method, or field may be referenced in a safety-critical application. This means that an application at Level n will be permitted only to use items labelled with @SCJAllowed(n) or lower. This also means that an application at Level n can be executed only by an implementation at Level n or higher.

Additionally, there are a number of annotations that restrict application code in several ways that enable or enhance static analyzability. For example, only methods that are annotated @SCJMayAllocate(CurrentContext) may contain expressions that result in memory allocation in the current memory area, and such methods may not contain expressions that result in memory allocation in any other memory area (e.g., Immortal memory, etc.). See Chapter 9 for details.

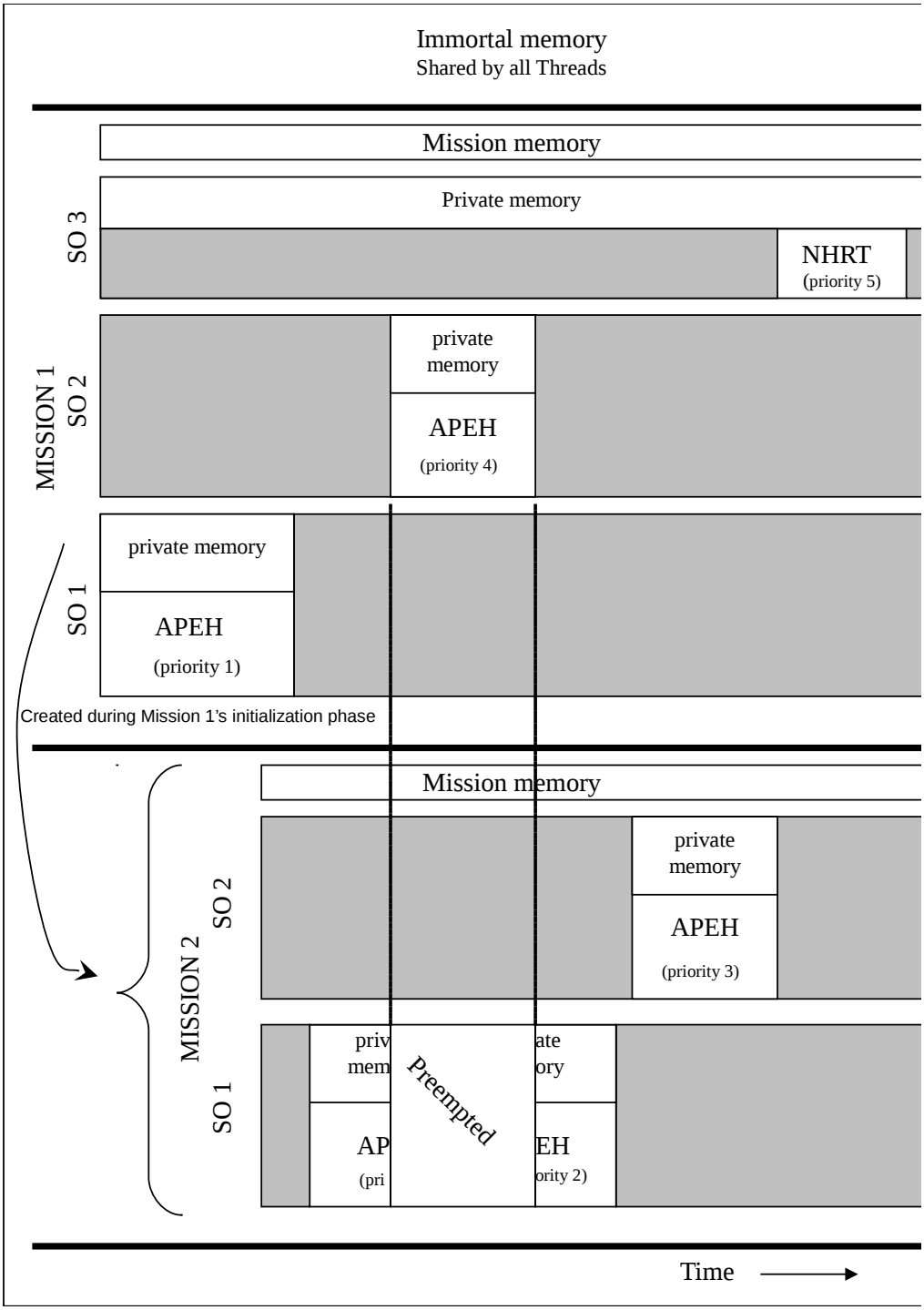

Figure 2.3: Level 2 [Nested Missions]

2.4 Use of Asynchronous Event Handlers

The RTSJ defines two mechanisms for real-time execution: the RealtimeThread and NoHeapRealtimeThread classes, which embody a programming style similar to java.lang.Thread for concurrent programming, and the AsynchronousEventHandler class, which is event based. This specification does not require the presence of a garbage-collected heap, thus the use of RealtimeThread is prohibited. To facilitate analyzability, this specification supports the following at each level:

- Level 0: PeriodicEventHandlers.
- Level 1: PeriodicEventHandlers AperiodicEventHandlers, and OneShotEventHanders.
- Level 2: PeriodicEventHandlers, AperiodicEventHandlers, OneShotEventHanders, and ManagedThreads.

The classes PeriodicEventHandler and AperiodicEventHandler are defined by this specification. The PeriodicEventHandler class is essentially the same as the AperiodicEventHandler class except that the PeriodicEventHandler class is defined with dispatching parameters that result in a periodic execution based on a timer. The application programmer establishes a periodic activity by extending the class PeriodicEventHandler, overriding the handleAsyncEvent method to perform the processing needed at each release, and constructing an instance with the desired priority and release parameters. This is different from the semantics of the AsynchronousEventHandler defined in the RTSJ, which requires associating an AsynchronousEventHandler object with a periodic timer if periodic dispatching is desired.

Sporadic AsynchronousEventHandler objects are not provided because their management would require the implementation to monitor minimal interarrival times for asynchronous events. It was determined that this would add excessive complexity with a resulting impact on safety-critical certifiability. This means that the application designer will need to carefully constrain its asynchronous event arrivals to avoid unbounded computation that can severely compromise the ability of the application to meet its time constraints.

2.5 Development vs. Deployment Compliance

As previously described in this specification, in a safety-critical application, certification requirements impose very stringent constraints on both the Java implementation and the application. This specification describes many syntactic and semantic limitations intended to enable the development of certifiable implementations and applications with a maximum level of portability across both development and execution platforms.

This specification requires that a conforming implementation provide all of the interfaces, operating according to the specified semantics, to every conforming application.

These requirements are to be strictly imposed on implementations that are capable of deployment into safety-critical environments. In contrast, for implementations usable only during development, while it is preferable for these requirements to be imposed, a limited number of deviations from this specification are explicitly permitted. These deviations are:

- Implementations running on an RTSJ-compliant JVM are permitted to support RTSJ interfaces that are not enumerated by this specification. Applications conforming to this specification are not permitted to make use of these interfaces.
- Implementations running on an RTSJ-compliant JVM must support the interfaces supporting Priority Ceiling Emulation (PCE), but are not required to support the PCE semantics if the underlying RTSJ implementation does not support PCE. Applications conforming to this specification may not execute exactly as expected because of the use of Priority Inheritance semantics for synchronization rather than Priority Ceiling Emulation as required by this specification.

2.6 Verification of Safety Properties

This specification omits a large number of RTSJ and other Java capabilities, such as dynamic class loading, in its effort to create a subset of Java capabilities that can be certified under a variety of safety standards such as DO-178C.

However, it is clear that no specification can, by itself, ensure the complete absence of unsafe operations in a conforming application. As a result, a further recommendation for an implementation is to provide a variety of pre-deployment analysis tools that can ensure the absence of certain unsafe operations. While this specification does not define particular analysis tools, it is extremely important that applications be certifiably free of memory reference errors. When analysis tools provided with an implementation are able to certify freedom of memory reference errors, the implementation need not provide run-time checking for such errors.

Chapter 3

Mission Life Cycle

This chapter describes the *Mission* concept, how it works, how it starts and stops, and how it is used in an SCJ application. In this chapter, the sections Semantics and Requirements, Level Considerations, and API are normative. The Overview and Rationale sections are not normative but are provided to improve understanding of the normative sections.

3.1 Overview

The mission concept is central to the design of SCJ. Whereas conventional Java provides various mechanisms to enforce encapsulation of data and functional behavior, SCJ's mission concept adds the ability to encapsulate multiple threads of control, identified as ManagedSchedulables, with accompanying data structures and functional behavior within a mission.

Every SCJ application is comprised of a sequence of missions containing at least one mission, with each mission representing a different operational phase. For example, an airplane's control software might be structured as a sequence of four missions supporting the taxi, takeoff, cruise, and land phases of operation.

It is also possible to structure an SCJ application as multiple concurrently active missions. Such a SCJ mission concept allows large and complex applications to be divided into multiple active components that can be developed, certified, and maintained largely in isolation of each other. For example, an aircraft's flight control software might be hierarchically decomposed into missions that independently focus on radio communications, global positioning, navigation and routing, collision avoidance, coordination with air traffic control, and automatic pilot operation.

An SCJ mission has three phases: an *initialization phase*, an *execution phase*, and a *clean up phase* as illustrated in Figure 3.1. This supports a common design pattern

for safety-critical systems in which shared data structures are allocated during the initialization phase before the system becomes active.

Every SCJ application runs under the direction of an application-provided mission sequencer. The mission sequencer organizes a sequence of one or more application-defined missions, each of which is an object of a class extending the Mission class. Infrastructure invokes the mission sequencer's getNextMission method both to select the initial mission and to select which mission to run next when a running mission terminates. This is illustrated in Figure 3.1.

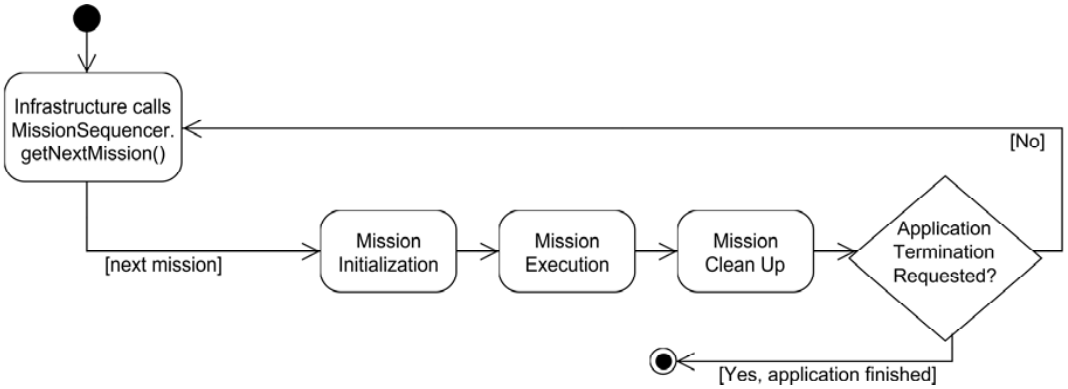

Figure 3.1: Safety-Critical Application Phases

Because the initial mission sequencer does not belong to any mission, the corresponding MissionSequencer object resides in the ImmortalMemory.

The mission sequencer is represented by a subclass of MissionSequencer, which extends ManagedEventHandler and implements ManagedSchedulable, both of which are defined further in Chapter 4. While SCJ missions at all Levels are comprised of ManagedSchedulables, one of the key differentiating features of Level 2 missions is their ability to run inner-nested mission sequencers concurrently with other ManagedSchedulable objects within the same mission, permitting the application to run multiple missions concurrently.

3.1.1 Application Initialization

An SCJ application is represented by an application-defined implementation of the Safelet interface. The application class that implements Safelet provides definitions of the immortalMemorySize, globalBackingStoreSize, handleStartupError, initializeApplication, and getSequencer methods.

The infrastructure invokes immortalMemorySize, globalBackingStoreSize, initializeApplication, and getSequencer in this order with ImmortalMemory as the current alloca-

tion context. If the infrastructure's call to immortalMemorySize and/or globalBackingStoreSize results in a request for more memory than is available, the infrastructure shall call handleStartupError to determine whether the application shall be immediately halted, or whether these calls shall be repeated, making it possible for an application to continue in a degraded configuration such as a partial memory failure.

The application may allocate objects in immortal memory within the initializeApplication method. The getSequencer method returns a reference to the MissionSequencer that oversees execution of the application.

3.1.2 Mission Initialization

The infrastructure invokes the initialize method associated with each mission. The initialize method, which is written by the application developer, instantiates all of the ManagedSchedulable and ManagedInterruptServiceRoutine objects that are intended to run as part of this mission. For each instantiated object, the application code invokes its corresponding register method to make it active during the mission. Only those ManagedSchedulables and ManagedInterruptServiceRoutine objects registered during initialize() will be available for release. Furthermore, it allocates and initializes objects that will be shared among these objects.

All ManagedSchedulable and ManagedInterruptServiceRoutine objects shall be allocated in the mission memory area of the mission to which they belong. This shall be checked at run time by the register method that shall throw an IllegalArgumentException if it is violated.

3.1.3 Mission Execution

Upon return from the initialize method, the SCJ infrastructure activates the objects that were registered during initialization.

For each managed schedulable object (and optionally for interrupt handlers), the SCJ infrastructure provides a private memory area which serves as the default memory area to hold temporary memory allocations. Each private memory area may hold additional inner-nested private memory areas to hold temporary objects that have shorter lifetimes than the duration of each release. During execution, objects may also be allocated in outer memory areas such as ImmortalMemory or in outer-nested mission memory areas.

Mission execution continues until each of the mission's managed schedulable objects terminates. A ManagedThread terminates by simply returning from its run() method. The only way to terminate ManagedEventHandlers is to terminate the mission by in-

voking the mission's requestTermination method. In this way, each ManagedEventHandler object will terminate following completion of its current release.

3.1.4 Mission Clean Up

The application defines a cleanUp method for each mission. The SCJ infrastructure invokes the mission cleanUp method after all of the managed schedulable objects registered with this mission have terminated their execution and each of their cleanUp methods have been invoked.

The cleanUp method can be used to free resources and to restore system state. For example, an application-defined cleanUp method may close files that had been opened during mission initialization or execution, and it might power down a device that was being controlled by the mission.

The application's mission cleanUp method determines whether it is appropriate for the MissionSequencer to continue with the execution of its sequence of missions. The cleanUp method returns true to indicate that the sequencer should continue or false to indicate that the MissionSequencer itself should terminate.

3.2 Semantics and Requirements

An application consists of one or more missions executed sequentially or concurrently, as initiated by a application-defined implementation of the Safelet interface. Each Mission has its own mission memory which holds objects representing the Mission state. The ManagedSchedulable objects that comprise the Mission generally communicate with each other by modifying shared objects that reside within the mission memory.

An application's execution consists of several steps, as outlined below.

3.2.1 Class Initialization

Class initialization is performed by the infrastructure on all application classes which shall have been placed in immortal memory by the implementation. The SCJ infrastructure shall initialize all of the classes that comprise the application in an order determined by a topological sort of class interdependencies before performing Safelet initialization. To ensure successful class initialization, an SCJ application shall have no cyclic dependencies among its class initialization methods. For purposes of the dependency analysis, it is strongly recommended that conforming implementations

should provide a *Checker* utility to enforce the following rules in a static analysis of the application's bytecodes:

- Virtual method invocations are prohibited from within a <clinit>[1] method unless data-flow analysis of the <clinit> method alone (without any analysis of code outside this <clinit> method) is able to prove the concrete type of the virtual method invocation's target object. Specifically, virtual method invocation is allowed only if every reaching definition of the invocation target object is the result of a new object allocation for the same concrete type.
- If an instance of some other class is allocated from within this <clinit> method, the class initialization for this class is considered to depend on class initialization of the allocated class.
- If an instance of a static field belonging to some other class is read or written from within this <clinit> method, the class initialization for this class is considered to depend on class initialization of that other class.
- If an instance or static method belonging to some other class is invoked from within this <clinit> method, the class initialization for this class is considered to depend on class initialization of that other class.
- The analysis of dependencies among class initialization methods shall not depend on control-flow analysis. While, in general, the problem of determining all class dependencies is intractable, a safe analysis can be successfully performed using more or less sophisticated heuristic algorithms. For example, a more sophisticated analysis might be able to prove that certain dependencies of one class on other classes reside within code that is never executed (i.e. "dead code"). Since a dependency analysis ignores control-flow considerations, dead-code dependencies identifiable by control-flow analysis shall be treated as if they were actual dependencies.
- The analysis of class initialization dependencies is performed on bytecode. If a Java source compiler recognizes and removes dead code from the bytecode, any dependencies in the eliminated dead code shall not be considered in the dependency analysis.
- The analysis of cyclic dependencies for a class does not forbid dependencies on self. It is common for <clinit> methods to make reference to the fields and methods of the class being initialized. It is the application programmer's responsibility to avoid unresolved dependencies within the class that is being initialized.

[1] The <clinit> method is a class initialization method created by the Java compiler when the class is compiled

3.2.2 Safelet Initialization

An implementation-specific initialization thread running at an implementation-specific thread priority takes responsibility for running Safelet specific code. An SCJ - compliant implementation of this startup thread shall implement the semantics in this stated order.

1. The Safelet object is allocated within the ImmortalMemory area.

2. The Safelet's immortalMemorySize method is invoked to determine the desired size of the ImmortalMemory. If the actual size of the remaining ImmortalMemory is smaller than the value returned from immortalMemorySize, the infrastructure shall call the handleStartupError method to determine whether the application should be immediately halted.

3. The Safelet's globalBackingStoreSize method is invoked to determine the desired size of the backing store required for all managed memory areas. If the actual size of the remaining backing store is smaller than the value returned from immortalMemorySize, the infrastructure shall call the handleStartupError method to determine whether the application should be immediately halted.

4. Infrastructure invokes the Safelet's initializeApplication method to allow the application to allocate global data structures in the ImmortalMemory.

5. Infrastructure invokes the Safelet's getSequencer method, with the ImmortalMemory area as the current allocation context. The value returned represents the MissionSequencer that runs this application. If null is returned, the application immediately aborts.

The getSequencer method is not allowed to invoke ManagedMemory.enterPrivateMemory. This is enforced with a run-time check. Any attempt to do so will abort by throwing an IllegalStateException.

Exceptions generated by getSequencer that are not handled within its implementation shall follow the rules for propagation and handling described in Chapter 11 for application methods.

It is implementation-defined how much total memory is available within the SCJ run-time environment to satisfy the combined requests for immortal memory, scoped memory areas, and stack memory generated by the application. Whether the memory used to satisfy each scoped memory and stack memory request is subject to fragmentation is also implementation defined.

3.2.3 MissionSequencer Execution

The SCJ infrastructure shall perform as if the following sequence were performed in this stated order. At any point during this process, if the infrastructure determines that (1) the MissionSequencer has been requested to terminate by an outer level mission, and (2) a Mission's initialize method has returned, and (3) the mission has not yet been started, then the infrastructure shall abandon the mission immediately and its cleanUp method shall be called.

1. Infrastructure code creates and starts up the MissionSequencer. The memory resources specified by the ScopeParameters argument to the MissionSequencer's constructor are set aside at the time that the infrastructure starts the MissionSequencer.

2. Next, infrastructure code releases the MissionSequencer.

3. In the case that this is the outer-most MissionSequencer associated with a Safelet, the implementation shall behave as if the Safelet initialization thread blocks itself and does not run throughout execution of the SCJ application.[2]

4. In the case that a MissionSequencer nests within a Level 2 Mission, the MissionSequencer must be registered during execution of that Mission's initialize code. Its memory resource requirements are specified by the ScopeParameters argument to its MissionSequencer constructor. These resources are reserved for execution of the MissionSequencer at the time the MissionSequencer is started, following return from the enclosing Mission's initialize method.

5. When the MissionSequencer begins to execute, it instantiates a mission memory object to hold data corresponding to the missions that are to be executed by this MissionSequencer. The backing store associated with this mission memory object is initially sized to represent all of the remaining backing store memory specified by the ScopeParameters of this MissionSequencer.

6. Next, the MissionSequencer enters the newly created mission memory area and invokes its own getNextMission method to obtain a reference to the first mission to be executed by this MissionSequencer. The getNextMission method, which is written by the application developer, may allocate and return a new Mission object in the mission memory area, or it may return a Mission object that resides in some memory area in an outer scope.

[2] A possible implementation-dependent optimization may use the same thread to perform Safelet initialization and the initial MissionSequencer's event handling.

7. When a Mission object is returned from getNextMission, the MissionSequencer invokes its missionMemorySize method and truncates the current mission memory area to the size returned from the missionMemorySize method. If the value returned is larger than the size of the current mission memory, the MissionSequencer aborts the current mission, exits the current mission memory, reclaiming the memory of all objects allocated within it, and endeavors to replace the current mission with a new mission by reinvoking its getNextMission method.

8. When the size of mission memory is truncated, the surplus memory that had previously been part of the current mission memory area's backing store is returned to the pool of backing store memory available for schedulable objects of this mission or with its inner-nested missions.

9. After successfully resizing mission memory, the MissionSequencer invokes the selected Mission object's initialize method. If upon return from initialize, the MissionSequencer has been requested to terminate, the MissionSequencer shall abandon execution of the mission and shall call the mission cleanUp method. It is the application's responsibility to coordinate the initialize and mission cleanUp code so that any state modified during a partial initialization is properly restored.

10. If upon return from initialize, the MissionSequencer has not been requested to terminate, the MissionSequencer starts all of the managed schedulable objects and interrupt handlers that were registered by the initialize method. The MissionSequencer then awaits mission termination.

11. After the MissionSequencer terminates, the infrastructure shall call the Safelet's cleanUp method so the safelet can perform logging or another application-defined operations.

In general, the management of memory to satisfy the ScopeParameters specified by the arguments of a MissionSequencer's constructor is implementation-defined. The memory management technique used for the initial MissionSequencer's ScopeParameters request may be different from the memory management technique used for MissionSequencers nested within a Level 2 mission.

3.2.4 Mission Execution

Each mission shall execute as if the following detailed steps that comprise mission execution are performed in this stated order:

1. With mission memory as the current allocation context, the MissionSequencer invokes the Mission's initialize method, which is written by the application developer. Within the initialize method, application code registers all of the ManagedSchedulable and ManagedInterruptServiceRoutine objects that will run as part of this mission. A ScopeParameters object shall be associated with each ManagedSchedulable at construction time. This ScopeParameters object shall describe the resources required for execution of the corresponding ManagedSchedulable. Reservation of the requested resources is made at the time the ManagedSchedulable object is constructed. It is common, but not necessary, for all of a mission's ManagedSchedulable objects to be allocated within mission memory by the initialize method. If a ManagedSchedulable to be associated with this mission is not allocated in this mission's mission memory, then it must reside in an outer memory area.

2. The initialize method may also allocate mission-relevant data structures in mission memory. These data structures may be shared between the managed schedulable objects of this mission. The initialize method may also use nested private memory areas to hold temporary objects relevant to its computations.

3. In a Level 0 application, upon return from initialize, the infrastructure creates an array in mission memory representing all of the ManagedSchedulable objects that were registered by the initialize method and passes this array to the mission's CyclicExecutive.getSchedule method. To provide predictable application behavior, the entries within this array are sorted in the same order that the initialize method registered the objects. This array is used by the infrastructure to define the sequence and timing of each ManagedSchedulable object in its correct sequence during mission execution. If CyclicExecutive.getSchedule returns null or aborts by throwing an exception, control proceeds directly to execution of the Mission object's cleanUp method without executing any of the code associated with this mission's registered ManagedSchedulables. Otherwise, a cyclic executive is started that runs according to the schedule.

4. For each ManagedSchedulable object, the infrastructure reserves the memory resources requested by the corresponding ManagedSchedulable object's ScopeParameters object. The backing store for each memory resource is obtained by setting aside portions of the backing store memory that had been previously associated with the MissionSequencer. It is not required that the backing store memories for each ManagedSchedulable be represented by contiguous memory. However, it is required that the memory behave as if it were contiguous memory. Subsequent instantiations of private memory areas shall not fail due to fragmentation.

5. The MissionSequencer then blocks waiting for the Mission's execution to ter-

minate. Once termination has been requested, the MissionSequencer shall complete the Mission termination sequence. If this termination sequence requires the use of implementation-defined resources, they must be accounted for by the application to provide highly predictable timing behavior. The termination sequence shall include calling the signalTermination method of each of the ManagedSchedulables associated with the current mission.

6. When the MissionSequencer detects that the Mission's mission phase has terminated, by confirming that all of the mission's ManagedSchedulable objects have terminated, it arranges to execute the cleanUp method of each of those ManagedSchedulables.

7. When a cleanUp method is called, a private memory area shall be provided for its use, and shall be the current allocation context. If desired, the cleanUp method may also create one or more new private memory areas. All memory allocated to each of the ManagedSchedulables shall be available to be freed when its cleanUp method returns. If an exception is thrown in a cleanUp method and is not caught in the method, it shall be caught and ignored by the MissionSequencer.

8. After the cleanUp methods for all of the mission's ManagedSchedulable objects have been executed, the MissionSequencer invokes the Mission's cleanUp method.

9. After the mission finishes its cleanUp code, control exits its mission memory area, releasing all of the objects that had been allocated within that mission memory, including the Mission object itself if it was allocated within the mission memory area.

If an exception is propagated from a call to initialize or cleanUp, it is caught and ignored. If the exception was propagated from initialize, the MissionSequencer shall run the next mission.

3.3 Level Considerations

3.3.1 Level 0

A Level 0 application shall implement Safelet<CyclicExecutive>. The getSequencer method of Safelet<CyclicExecutive> is declared to return a MissionSequencer<CyclicExecutive> object. The getNextMission method of MissionSequencer<Cyclic Executive> is declared to return a CyclicExecutive object. Thus, the type system enforces that

a Level 0 application is comprised only of CyclicExecutive missions. This is important because the SCJ infrastructure requires that a CyclicSchedule be associated with each Mission in the Level 0 application.

The CyclicExecutive subclass must implement the CyclicExecutive.getSchedule method. This method returns a reference to a CyclicSchedule object which represents the static cyclic schedule for the periodic event handlers associated with this CyclicExecutive object.

3.3.2 Level 1

A Level 1 application shall implement Safelet<Mission>. In particular, the application needs to provide a getSequencer method to return the mission sequencer of the application.

3.3.3 Level 2

A Level 2 application shall implement Safelet<Mission>, similar to a Level 1 application. An enhanced capability of Level 2 applications is the option to register ManagedThread objects and inner-nested MissionSequencer objects during execution of a Mission object's initialize method.

3.4 API

This section provides the detailed javadoc descriptions of relevant class APIs. The UML class diagram shown in Figure 3.2 illustrates the relationships between the classes described in this chapter.

3.4.1 javax.safetycritical.Safelet

Declaration

```
@SCJAllowed
public interface Safelet
```

Description

A safety-critical application consists of one or more missions, executed concurrently or in sequence. Every safety-critical application must implement Safelet

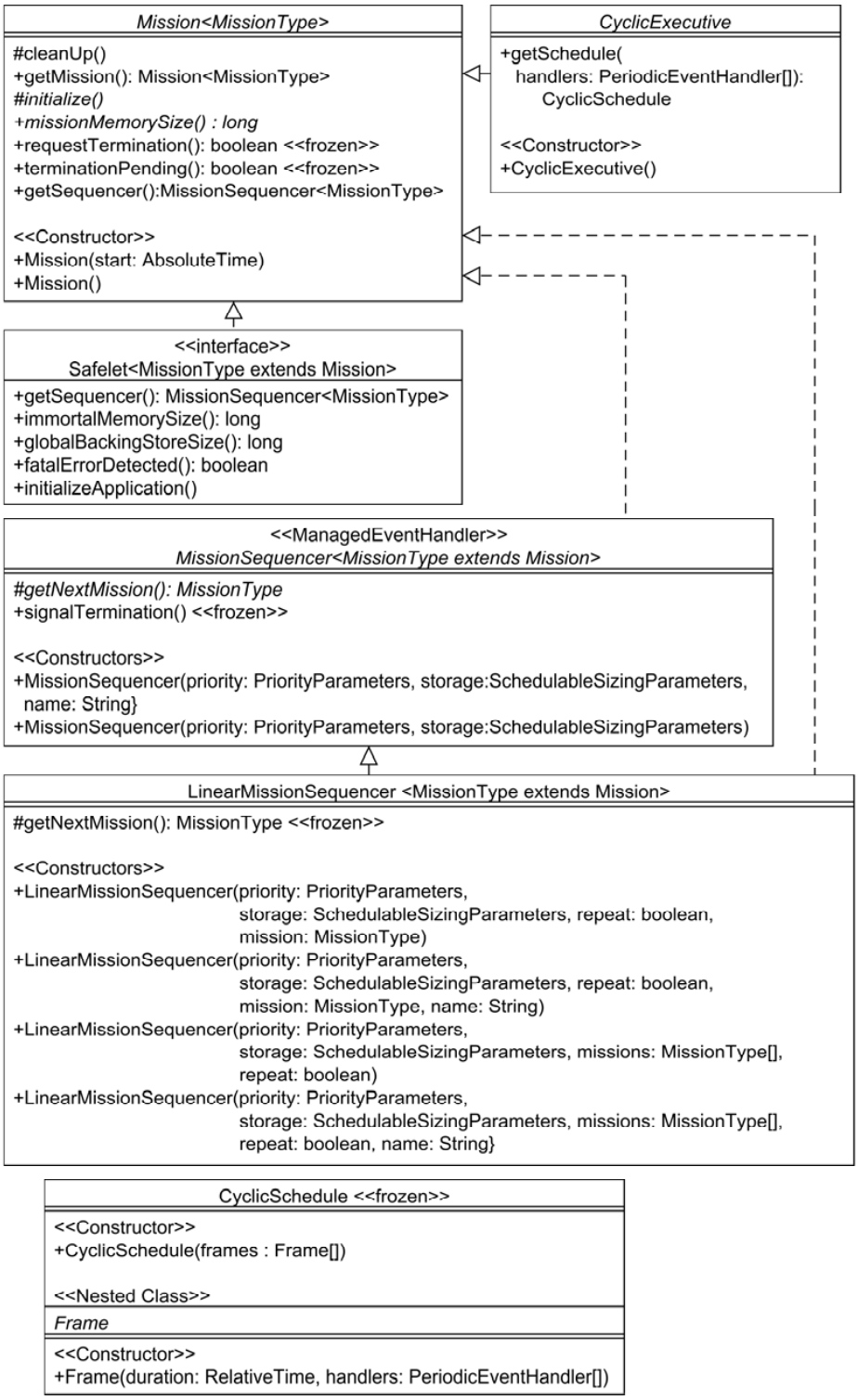

Figure 3.2: UML class diagram of classes related to mission life cycle

which identifies the outer-most MissionSequencer. This outer-most MissionSequencer runs the sequence of missions that comprise this safety-critical application.

The mechanism used to identify the Safelet to a particular SCJ environment is implementation defined.

Fields

```
@SCJAllowed
public static final long INSUFFICIENT_BACKING_STORE
```

```
@SCJAllowed
public static final long INSUFFICIENT_IMMORTAL_MEMORY
```

Methods

```
@SCJAllowed(javax.safetycritical.annotate.Level.SUPPORT)
@SCJPhase({javax.safetycritical.annotate.Phase.CLEANUP})
@SCJMayAllocate({javax.safetycritical.annotate.AllocationContext.CURRENT})
@SCJMaySelfSuspend(false)
public void cleanUp( )
```

Called by the infrastructure after termination of the MissionSequencer for this Safelet.

```
@SCJAllowed(javax.safetycritical.annotate.Level.SUPPORT)
@SCJPhase({javax.safetycritical.annotate.Phase.STARTUP})
@SCJMayAllocate({javax.safetycritical.annotate.AllocationContext.CURRENT})
@SCJMaySelfSuspend(false)
public javax.safetycritical.MissionSequencer getSequencer( )
```

The infrastructure invokes getSequencer to obtain the MissionSequencer object that oversees execution of missions for this application. The returned MissionSequencer resides in immortal memory.

returns the MissionSequencer that oversees execution of missions for this application.

```
@SCJAllowed(javax.safetycritical.annotate.Level.SUPPORT)
@SCJPhase({javax.safetycritical.annotate.Phase.STARTUP})
@SCJMayAllocate({})
@SCJMaySelfSuspend(false)
public long globalBackingStoreSize( )
```

returns the amount of additional backing store memory that must be available for managed memory areas. If the amount of remaining memory is less than this requested size, the infrastructure shall call the handleStartupError() method to determine whether the application should be immediately halted.

```
@SCJAllowed(javax.safetycritical.annotate.Level.SUPPORT)
@SCJPhase({javax.safetycritical.annotate.Phase.STARTUP})
@SCJMayAllocate({javax.safetycritical.annotate.AllocationContext.CURRENT})
@SCJMaySelfSuspend(false)
public boolean handleStartupError(int cause, long val)
```

> Called during startup by the infrastructure if the infrastructure detects the presence of a fatal startup error allocating memory or for any other implementation defined reason. This method returns a boolean indication whether it intends for the infrastructure to immediately halt execution, or whether it intends for the infrastructure to retry the failed allocation request. This method makes it possible for an application to attempt to execute in a degraded mode in the event of certain types of failures, such as a partial memory failure.

returns True if the infrastructure should immediately halt as a result of detecting the fatal startup error. If False is returned, the infrastructure should repeat its calls to immortalMemorySize() and globalBackingStoreSize(), providing the application the ability to reconfigure itself, if possible, to work around the fatal startup error.

cause — Identifies the condition that caused the infrastructure to call this method. If cause = INSUFFICIENT_IMMORTAL_MEMORY, the amount of available memory is insufficient for the immortal memory requested by the previous call to immortalMemorySize(). If cause = INSUFFICIENT_BACKING_STORE, the amount of available memory is insufficient for the backing store memory requested by the previous call to globalBackingStoreSize(). If cause has any other value, its meaning is implementation defined.

val — If cause = INSUFFICIENT_IMMORTAL_MEMORY, val contains the shortfall in available memory for the immortal memory requested by the previous call to immortalMemorySize(). If cause = INSUFFICIENT_BACKING_STORE, val contains the shortfall in available memory for the backing store memory requested by the previous call to globalBackingStoreSize(). If cause has any other value, the meaning of val is implementation defined.

```
@SCJAllowed(javax.safetycritical.annotate.Level.SUPPORT)
@SCJPhase({javax.safetycritical.annotate.Phase.STARTUP})
@SCJMayAllocate({})
@SCJMaySelfSuspend(false)
public long immortalMemorySize( )
```

returns the amount of additional immortal memory that must be available for allocations to be performed by this application. If the amount of remaining memory is less than this requested size, the infrastructure shall call the handleStartupError() method to determine whether the application should be immediately halted.

```
@SCJAllowed(javax.safetycritical.annotate.Level.SUPPORT)
@SCJPhase({javax.safetycritical.annotate.Phase.STARTUP})
@SCJMayAllocate({javax.safetycritical.annotate.AllocationContext.CURRENT})
@SCJMaySelfSuspend(true)
public void initializeApplication( )
```

The infrastructure shall invoke initializeApplication in the allocation context of immortal memory. The application can use this method to allocate data structures in immortal memory. This method shall be called exactly once by the infrastructure.

3.4.2 javax.safetycritical.MissionSequencer

Declaration

```
@SCJAllowed
 public abstract class MissionSequencer extends
     javax.safetycritical.ManagedEventHandler
```

Description

A MissionSequencer oversees a sequence of Mission executions. The sequence may include interleaved execution of independent missions and repeated executions of missions.

As a subclass of ManagedEventHandler, MissionSequencer's execution priority and memory budget are specified by constructor parameters.

This MissionSequencer executes vendor-supplied infrastructure code which invokes user-defined implementations of getNextMission, Mission.initialize, and Mission.cleanUp. During execution of a mission, the MissionSequencer remains blocked waiting for the mission to terminate. An invocation of signalTermination will unblock it to invoke the running mission's requestTermination method.

Constructors

```
@SCJAllowed
@SCJPhase({
   javax.safetycritical.annotate.Phase.STARTUP,
   javax.safetycritical.annotate.Phase.INITIALIZATION})
@SCJMaySelfSuspend(false)
```

```
@SCJMayAllocate({
    javax.safetycritical.annotate.AllocationContext.CURRENT,
    javax.safetycritical.annotate.AllocationContext.INNER,
    javax.safetycritical.annotate.AllocationContext.OUTER})
public MissionSequencer(PriorityParameters priority,
  ScopeParameters storage,
  ConfigurationParameters config,
  String name)
  throws java.lang.IllegalStateException
```

Construct a MissionSequencer object to oversee a sequence of mission executions.

priority — The priority at which the MissionSequencer executes.

storage — specifies the ScopeParameters for this handler

config — specifies the ConfigurationParameters for this handler

name — The name by which this MissionSequencer will be identified.

Throws IllegalStateException if invoked in an inappropriate phase.

```
@SCJAllowed
@SCJPhase({
    javax.safetycritical.annotate.Phase.STARTUP,
    javax.safetycritical.annotate.Phase.INITIALIZATION})
@SCJMaySelfSuspend(false)
@SCJMayAllocate({
    javax.safetycritical.annotate.AllocationContext.CURRENT,
    javax.safetycritical.annotate.AllocationContext.INNER,
    javax.safetycritical.annotate.AllocationContext.OUTER})
public MissionSequencer(PriorityParameters priority,
  ScopeParameters storage,
  ConfigurationParameters config)
  throws java.lang.IllegalStateException
```

This constructor behaves the same as calling MissionSequencer(priority, storage, config, null).

Methods

```
@SCJAllowed(javax.safetycritical.annotate.Level.SUPPORT)
@SCJPhase({
    javax.safetycritical.annotate.Phase.STARTUP,
    javax.safetycritical.annotate.Phase.INITIALIZATION,
    javax.safetycritical.annotate.Phase.CLEANUP})
@SCJMaySelfSuspend(false)
@SCJMayAllocate({
```

```
    javax.safetycritical.annotate.AllocationContext.CURRENT,
    javax.safetycritical.annotate.AllocationContext.INNER,
    javax.safetycritical.annotate.AllocationContext.OUTER})
protected abstract javax.safetycritical.Mission getNextMission( )
```

This method is called by infrastructure to select the initial mission to execute, and subsequently, each time one mission terminates, to determine the next mission to execute.

Prior to each invocation of getNextMission, infrastructure initializes and enters the mission memory allocation area. The getNextMission method may allocate the returned mission within this mission memory area, or it may return a reference to a Mission object that was allocated in some outer-nested mission memory area or in the ImmortalMemory area.

returns the next mission to run, or null if no further missions are to run under the control of this MissionSequencer.

```
@SCJAllowed(javax.safetycritical.annotate.Level.SUPPORT)
@Override
@SCJMaySelfSuspend(true)
@SCJMayAllocate({
    javax.safetycritical.annotate.AllocationContext.CURRENT,
    javax.safetycritical.annotate.AllocationContext.INNER,
    javax.safetycritical.annotate.AllocationContext.OUTER})
@SCJPhase({
    javax.safetycritical.annotate.Phase.STARTUP,
    javax.safetycritical.annotate.Phase.INITIALIZATION,
    javax.safetycritical.annotate.Phase.RUN,
    javax.safetycritical.annotate.Phase.CLEANUP })
public final void handleAsyncEvent( )
```

This method is used in the implementation of SCJ infrastructure. The method is not to be invoked by application code and it is not to be overridden by application code.

```
@SCJAllowed(javax.safetycritical.annotate.Level.SUPPORT)
@SCJMayAllocate({})
@SCJMaySelfSuspend(false)
@SCJPhase({
    javax.safetycritical.annotate.Phase.STARTUP,
    javax.safetycritical.annotate.Phase.INITIALIZATION,
    javax.safetycritical.annotate.Phase.RUN,
    javax.safetycritical.annotate.Phase.CLEANUP })
public final void signalTermination( )
```

Called by the infrastructure to indicate that the enclosing mission has been instructed to terminate.

The sole responsibility of this method is to call requestTermination on the currently running mission.

signalTermination will never be called by a Level 0 or Level 1 infrastructure.

3.4.3 javax.safetycritical.Mission

Declaration

```
@SCJAllowed
 public abstract class Mission extends java.lang.Object
```

Description

A Safety Critical Java application is comprised of one or more missions. Each mission is implemented as a subclass of this abstract Mission class. A mission is comprised of one or more ManagedSchedulable objects, conceptually running as independent threads of control, and the data that is shared between them.

Constructors

```
@SCJAllowed
@SCJPhase({
   javax.safetycritical.annotate.Phase.STARTUP,
   javax.safetycritical.annotate.Phase.INITIALIZATION,
   javax.safetycritical.annotate.Phase.RUN})
@SCJMaySelfSuspend(false)
@SCJMayAllocate({
   javax.safetycritical.annotate.AllocationContext.CURRENT,
   javax.safetycritical.annotate.AllocationContext.INNER,
   javax.safetycritical.annotate.AllocationContext.OUTER})
public Mission(AbsoluteTime start)
```

Allocate and initialize data structures associated with a Mission implementation.

The constructor may allocate additional infrastructure objects within the same MemoryArea that holds the implicit this argument.

The amount of data allocated in the same MemoryArea as this by the Mission constructor is implementation-defined. Application code will need to know the amount of this data to properly size the containing scope.

start — an absolute time value at which the Mission's ManagedSchedulable objects will be released for the first time after mission initialization has been completed

unless they are delayed by their own start times. If start is null, or if start has already passed when the Mission's ManagedSchedulable objects become ready for release, they shall be released immediately.

```
@SCJAllowed
@SCJPhase({
    javax.safetycritical.annotate.Phase.STARTUP,
    javax.safetycritical.annotate.Phase.INITIALIZATION,
    javax.safetycritical.annotate.Phase.RUN})
@SCJMaySelfSuspend(false)
@SCJMayAllocate({
    javax.safetycritical.annotate.AllocationContext.CURRENT,
    javax.safetycritical.annotate.AllocationContext.INNER,
    javax.safetycritical.annotate.AllocationContext.OUTER})
public Mission( )
```

This constructor is equivalent to Mission(null).

Methods

```
@SCJAllowed(javax.safetycritical.annotate.Level.SUPPORT)
@SCJPhase({javax.safetycritical.annotate.Phase.CLEANUP})
@SCJMaySelfSuspend(true)
@SCJMayAllocate({
    javax.safetycritical.annotate.AllocationContext.CURRENT,
    javax.safetycritical.annotate.AllocationContext.INNER,
    javax.safetycritical.annotate.AllocationContext.OUTER})
protected boolean cleanUp( )
```

Cleans data structures and machine state upon termination of this Mission's run phase. Infrastructure running the controlling MissionSequencer invokes cleanUp after all ManagedSchedulables registered with this Mission have terminated, but before control leaves the corresponding mission memory area.

returns True to indicate that the mission sequencer shall continue with its sequence of missions, False to indicate that the mission sequence should be terminated and no further missions started. The default implementation of cleanUp returns True.

```
@SCJAllowed
@SCJMayAllocate({})
@SCJMaySelfSuspend(false)
@SCJPhase({
    javax.safetycritical.annotate.Phase.STARTUP,
    javax.safetycritical.annotate.Phase.INITIALIZATION,
    javax.safetycritical.annotate.Phase.RUN,
    javax.safetycritical.annotate.Phase.CLEANUP })
public static javax.safetycritical.Mission getMission( )
```

Obtain the current mission.

returns the instance of the Mission that is currently active.

If called during the initialization or clean up phase, getMission() returns the mission that is currently being initialized or cleaned up. If called during the run phase, getMission() returns the mission in which the currently executing ManagedSchedulable was registered. If called during the start up phase, getMission() returns null.

```
@SCJAllowed
@SCJPhase({
    javax.safetycritical.annotate.Phase.INITIALIZATION,
    javax.safetycritical.annotate.Phase.RUN,
    javax.safetycritical.annotate.Phase.CLEANUP})
@SCJMaySelfSuspend(false)
@SCJMayAllocate({})
public javax.safetycritical.MissionSequencer getSequencer( )
```

returns the MissionSequencer that is overseeing execution of this mission.

```
@SCJAllowed(javax.safetycritical.annotate.Level.SUPPORT)
@SCJPhase({
    javax.safetycritical.annotate.Phase.STARTUP,
    javax.safetycritical.annotate.Phase.INITIALIZATION})
@SCJMaySelfSuspend(true)
@SCJMayAllocate({
    javax.safetycritical.annotate.AllocationContext.CURRENT,
    javax.safetycritical.annotate.AllocationContext.INNER,
    javax.safetycritical.annotate.AllocationContext.OUTER})
protected abstract void initialize( )
```

Perform initialization of this Mission. The SCJ infrastructure calls initialize after the mission memory has been resized to match the size returned by Mission.missionMemorySize. Upon entry into initialize, the current allocation context is the mission memory area dedicated to this particular Mission.

A typical implementation of initialize instantiates and registers all ManagedSchedulable objects that constitute this Mission. The infrastructure enforces that ManagedSchedulables can only be instantiated and registered if the currently executing ManagedSchedulable is running a Mission.initialize method. The infrastructure arranges to begin executing the registered ManagedSchedulable objects associated with a particular Mission upon return from its initialize method.

Besides initiating the associated ManagedSchedulable objects, this method may also instantiate and/or initialize mission-level data structures. Objects shared between ManagedSchedulables typically reside within the corresponding mission memory scope, but may alternatively reside in outer-nested mission

memory or ImmortalMemory areas. Individual ManagedSchedulables can gain access to these objects either by supplying their references to the ManagedSchedulable constructors or by obtaining a reference to the currently running mission (from Mission.getMission), and accessing the fields or methods of this subclass.

```
@SCJAllowed(javax.safetycritical.annotate.Level.SUPPORT)
@SCJMayAllocate({})
@SCJPhase({javax.safetycritical.annotate.Phase.INITIALIZATION})
@SCJMaySelfSuspend(false)
public abstract long missionMemorySize( )
```

This method must be implemented by a safety-critical application. It is invoked by the SCJ infrastructure to determine the desired size of this Mission's mission memory area. When this method receives control, the mission memory area will include all of the backing store memory to be used for all memory areas. Therefore this method will not be able to create or call any methods that create any private memory areas. After this method returns, the SCJ infrastructure shall shrink the mission memory to a size based on the memory size returned by this method. This will make backing store memory available for the backing stores of the ManagedSchedulable objects that comprise this mission. Any attempt to introduce a new private memory area within this method will result in an OutOfMemoryError exception.

returns the required mission memory size.

```
@SCJAllowed
@SCJPhase({
   javax.safetycritical.annotate.Phase.INITIALIZATION,
   javax.safetycritical.annotate.Phase.RUN})
@SCJMayAllocate({})
@SCJMaySelfSuspend(false)
public final boolean requestTermination( )
```

This method requests termination of a mission. When this method is called, the infrastructure shall invoke signalTermination on each ManagedSchedulable object that is registered within this mission. Additionally, this method triggers the infrastructure to (1) disable all periodic event handlers associated with this Mission so that they will experience no further releases, (2) disable all AperiodicEventHandlers so that no further releases will be honored, (3) clear the pending event (if any) for each event handler (including any OneShotEventHandlers) so that the event handler can be effectively shut down following completion of any event handling that is currently active, (4) wait for all of the

ManagedSchedulable objects associated with this mission to terminate their execution, (5) invoke the ManagedSchedulable.cleanUp methods for each of the ManagedSchedulable objects associated with this mission, and (6) invoke the cleanUp method associated with this mission.

While many of these activities may be carried out asynchronously after returning from the requestTermination method, the implementation of requestTermination shall not return until all of the ManagedEventHandler objects registered with this Mission have been disassociated from this Mission so they will receive no further releases. Before returning, or at least before initialize for this same mission is called in the case that it is subsequently started, the implementation shall clear all mission state.

The first time this method is called during Mission execution, it shall return false to indicate that termination of this mission is not already in progress. Subsequent invocations of this method shall return true, and shall have no other effect.

returns false if the mission has not been requested to terminate already; otherwise returns true.

```
@SCJAllowed
@SCJPhase({
    javax.safetycritical.annotate.Phase.STARTUP,
    javax.safetycritical.annotate.Phase.INITIALIZATION,
    javax.safetycritical.annotate.Phase.RUN,
    javax.safetycritical.annotate.Phase.CLEANUP})
@SCJMayAllocate({})
@SCJMaySelfSuspend(false)
public final boolean terminationPending( )
```

Check whether the current mission is trying to terminate.

returns true if and only if this Mission's requestTermination method has been previously invoked.

3.4.4 javax.safetycritical.Frame

Declaration

```
@SCJAllowed
 public final class Frame extends java.lang.Object
```

Constructors

```
@SCJAllowed
@SCJPhase({javax.safetycritical.annotate.Phase.INITIALIZATION})
@SCJMaySelfSuspend(false)
@SCJMayAllocate({
   javax.safetycritical.annotate.AllocationContext.CURRENT,
   javax.safetycritical.annotate.AllocationContext.INNER,
   javax.safetycritical.annotate.AllocationContext.OUTER})
public Frame(RelativeTime duration, PeriodicEventHandler [] handlers)
```

Allocates and retains private shallow copies of the duration and handlers array within the same memory area as this. The elements within the copy of the handlers array are the exact same elements as in the handlers array. Thus, it is essential that the elements of the handlers array reside in memory areas that enclose this. Usually, this Frame object is instantiated within the mission memory area that corresponds to the Level 0 mission that is to be scheduled.

Within each execution frame of the CyclicSchedule, the PeriodicEventHandler objects represented by the handlers array will be released in the same order as they appear within this array.

3.4.5 javax.safetycritical.CyclicSchedule

Declaration

```
@SCJAllowed
 public final class CyclicSchedule extends java.lang.Object
```

Description

A CyclicSchedule object represents a time-driven sequence of firings for deterministic scheduling of periodic event handlers. The static cyclic scheduler repeatedly executes the firing sequence.

Constructors

```
@SCJAllowed
@SCJPhase({javax.safetycritical.annotate.Phase.INITIALIZATION})
@SCJMaySelfSuspend(false)
@SCJMayAllocate({
   javax.safetycritical.annotate.AllocationContext.CURRENT,
   javax.safetycritical.annotate.AllocationContext.INNER,
   javax.safetycritical.annotate.AllocationContext.OUTER})
public CyclicSchedule(CyclicSchedule.Frame [] frames)
  throws java.lang.IllegalArgumentException, java.lang.IllegalStateException
```

Construct a cyclic schedule by copying the frames array into a private array within the same memory area as this newly constructed CyclicSchedule object.

The frames array represents the order in which event handlers are to be scheduled. Note that some Frame entries within this array may have zero PeriodicEventHandlers associated with them. This would represent a period of time during which the CyclicExecutive is idle.

Throws IllegalArgumentException if any element of the frames array equals null or if the frames array is empty,

Throws IllegalStateException if invoked by a Level 1 a Level 2 application.

3.4.6 Class javax.safetycritical.CyclicExecutive

Declaration

```
@SCJAllowed
 public abstract class CyclicExecutive extends javax.safetycritical.Mission
```

Description

A CyclicExecutive represents a Level 0 mission. Every mission in a Level 0 application must be a subclass of CyclicExecutive.

Constructors

```
@SCJAllowed
@SCJPhase({
   javax.safetycritical.annotate.Phase.STARTUP,
   javax.safetycritical.annotate.Phase.INITIALIZATION})
@SCJMaySelfSuspend(false)
@SCJMayAllocate({
   javax.safetycritical.annotate.AllocationContext.CURRENT,
   javax.safetycritical.annotate.AllocationContext.INNER,
   javax.safetycritical.annotate.AllocationContext.OUTER})
public CyclicExecutive( )
```

Construct a CyclicExecutive object.

Methods

```
@SCJAllowed(javax.safetycritical.annotate.Level.SUPPORT)
@SCJPhase({
   javax.safetycritical.annotate.Phase.STARTUP,
   javax.safetycritical.annotate.Phase.INITIALIZATION})
@SCJMaySelfSuspend(false)
@SCJMayAllocate({
   javax.safetycritical.annotate.AllocationContext.CURRENT,
   javax.safetycritical.annotate.AllocationContext.INNER,
   javax.safetycritical.annotate.AllocationContext.OUTER})
public abstract javax.safetycritical.CyclicSchedule getSchedule(
  PeriodicEventHandler [] handlers)
```

Every CyclicExecutive shall provide its own cyclic schedule, which is represented by an instance of the CyclicSchedule class. Application programmers are expected to implement this method to provide a schedule that is appropriate for the mission.

Level 0 infrastructure code invokes the getSchedule method on the mission returned from MissionSequencer.getNextMission after invoking the mission's initialize method in order to obtain the desired cyclic schedule. Upon entry into the getSchedule method, this mission's mission memory area shall be the active allocation context. The value returned from getSchedule shall reside in the current mission's mission memory area area or in some enclosing scope.

Infrastructure code shall check that all of the PeriodicEventHandler objects referenced from within the returned CyclicSchedule object have been registered for execution with this Mission. If not, the infrastructure shall immediately terminate execution of this mission without executing any event handlers.

handlers — represents all of the handlers that have been registered with this Mission. The entries in the handlers array are sorted in the same order in which they were registered by the corresponding CyclicExecutive's initialize method. The infrastructure shall copy the information in the handlers array into its private memory, so subsequent application changes to the handlers array will have no effect.

returns the schedule to be used by the CyclicExecutive.

3.4.7 LinearMissionSequencer

Declaration

```
@SCJAllowed
 public class LinearMissionSequencer extends
     javax.safetycritical.MissionSequencer
```

Description

A LinearMissionSequencer is a MissionSequencer that serves the needs of a common design pattern in which the sequence of Mission executions is known prior to execution and all missions can be preallocated within an outer-nested memory area.

The parameter <M> allows application code to differentiate between LinearMissionSequencers that are designed for use in Level 0 vs. other environments. For example, a LinearMissionSequencer<CyclicExecutive> is known to only run missions that are suitable for execution within a Level 0 run-time environment.

Constructors

```
@SCJAllowed
@SCJPhase({javax.safetycritical.annotate.Phase.INITIALIZATION})
@SCJMaySelfSuspend(false)
@SCJMayAllocate({
    javax.safetycritical.annotate.AllocationContext.CURRENT,
    javax.safetycritical.annotate.AllocationContext.INNER,
    javax.safetycritical.annotate.AllocationContext.OUTER})
public LinearMissionSequencer(PriorityParameters priority,
  ScopeParameters storage,
  ConfigurationParameters config,
  Mission mission,
  boolean repeat,
  String name)
  throws java.lang.IllegalArgumentException, java.lang.IllegalStateException
```

Construct a LinearMissionSequencer object to oversee execution of a single mission m.

priority — The priority at which the MissionSequencer's bound thread executes.

storage — The memory resources to be dedicated to execution of this MissionSequencer's bound thread.

config — The configuration parameters to be dedicated to execution of this MissionSequencer's bound thread.

mission — The single mission that runs under the oversight of this LinearMissionSequencer.

repeat — When repeat is true, the specified mission shall be repeated indefinitely.

name — The name by which this LinearMissionSequencer will be identified in traces for use in debug or in toString.

Throws IllegalArgumentException if any of the arguments equals null.

```
@SCJAllowed
@SCJPhase({javax.safetycritical.annotate.Phase.INITIALIZATION})
@SCJMaySelfSuspend(false)
@SCJMayAllocate({
    javax.safetycritical.annotate.AllocationContext.CURRENT,
    javax.safetycritical.annotate.AllocationContext.INNER,
    javax.safetycritical.annotate.AllocationContext.OUTER})
public LinearMissionSequencer(PriorityParameters priority,
  ScopeParameters storage,
  ConfigurationParameters config,
  Mission mission,
  boolean repeat)
  throws java.lang.IllegalArgumentException, java.lang.IllegalStateException
```

This constructor behaves the same as calling LinearMissionSequencer(PriorityParameters, ConfigurationParameters, boolean, M, String) with the arguments (priority, storage, repeat, mission, null).

```
@SCJAllowed
@SCJPhase({javax.safetycritical.annotate.Phase.INITIALIZATION})
@SCJMaySelfSuspend(false)
@SCJMayAllocate({
   javax.safetycritical.annotate.AllocationContext.CURRENT,
   javax.safetycritical.annotate.AllocationContext.INNER,
   javax.safetycritical.annotate.AllocationContext.OUTER})
public LinearMissionSequencer(PriorityParameters priority,
  ScopeParameters storage,
  ConfigurationParameters config,
  Mission [] missions,
  boolean repeat,
  String name)
  throws java.lang.IllegalArgumentException, java.lang.IllegalStateException
```

Construct a LinearMissionSequencer object to oversee execution of the sequence of missions represented by the missions parameter. The LinearMissionSequencer runs the sequence of missions identified in its missions array exactly once, from low to high index position within the array. The constructor allocates a copy of its missions array argument within the current scope, so changes to .the missions array following construction will have no effect.

priority — The priority at which the MissionSequencer's bound thread executes.

storage — The memory resources to be dedicated to execution of this MissionSequencer's bound thread.

repeat — When repeat is true, the specified list of missions shall be repeated indefinitely.

missions — An array representing the sequence of missions to be executed under the oversight of this LinearMissionSequencer. Requires that the elements of the missions array reside in a scope that encloses the scope of this. The missions array itself may reside in a more inner-nested temporary scope.

name — The name by which this LinearMissionSequencer will be identified in traces for use in debug or in toString.

Throws IllegalArgumentException if any of the arguments equals null.

```
@SCJAllowed
@SCJPhase({javax.safetycritical.annotate.Phase.INITIALIZATION})
@SCJMaySelfSuspend(false)
```

```
@SCJMayAllocate({
    javax.safetycritical.annotate.AllocationContext.CURRENT,
    javax.safetycritical.annotate.AllocationContext.INNER,
    javax.safetycritical.annotate.AllocationContext.OUTER})
public LinearMissionSequencer(PriorityParameters priority,
  ScopeParameters storage,
  ConfigurationParameters config,
  Mission [] missions,
  boolean repeat)
  throws java.lang.IllegalArgumentException, java.lang.IllegalStateException
```

Same as LinearMissionSequencer(PriorityParameters, ConfigurationParameters, M[], boolean, String) with the arguments (priority, storage, missions, repeat, null).

Methods

```
@SCJAllowed(javax.safetycritical.annotate.Level.SUPPORT)
@SCJPhase({javax.safetycritical.annotate.Phase.INITIALIZATION})
@SCJMaySelfSuspend(false)
@SCJMayAllocate({})
@Override
protected final javax.safetycritical.Mission getNextMission( )
```

Returns a reference to the next Mission in the sequence of missions that was specified by the m or missions argument to this object's constructor.

See javax.safetycritical.MissionSequencer. getNextMission()

3.5 Application Initialization Sequence Diagram

A traditional standard edition Java application begins with execution of the static main method. The start up sequence for an SCJ application is a bit more complicated. Figure 3.3 uses a sample Level 1 application to provide an illustration of the interactions between the infrastructure and application code during the execution of an SCJ application.

3.6 Rationale

3.6.1 Loading and Initialization of Classes

With a traditional Java virtual machine, classes are generally loaded dynamically upon first access to the data or methods of the class. This implementation technique allows the Java virtual machine to start up more quickly, because it can begin

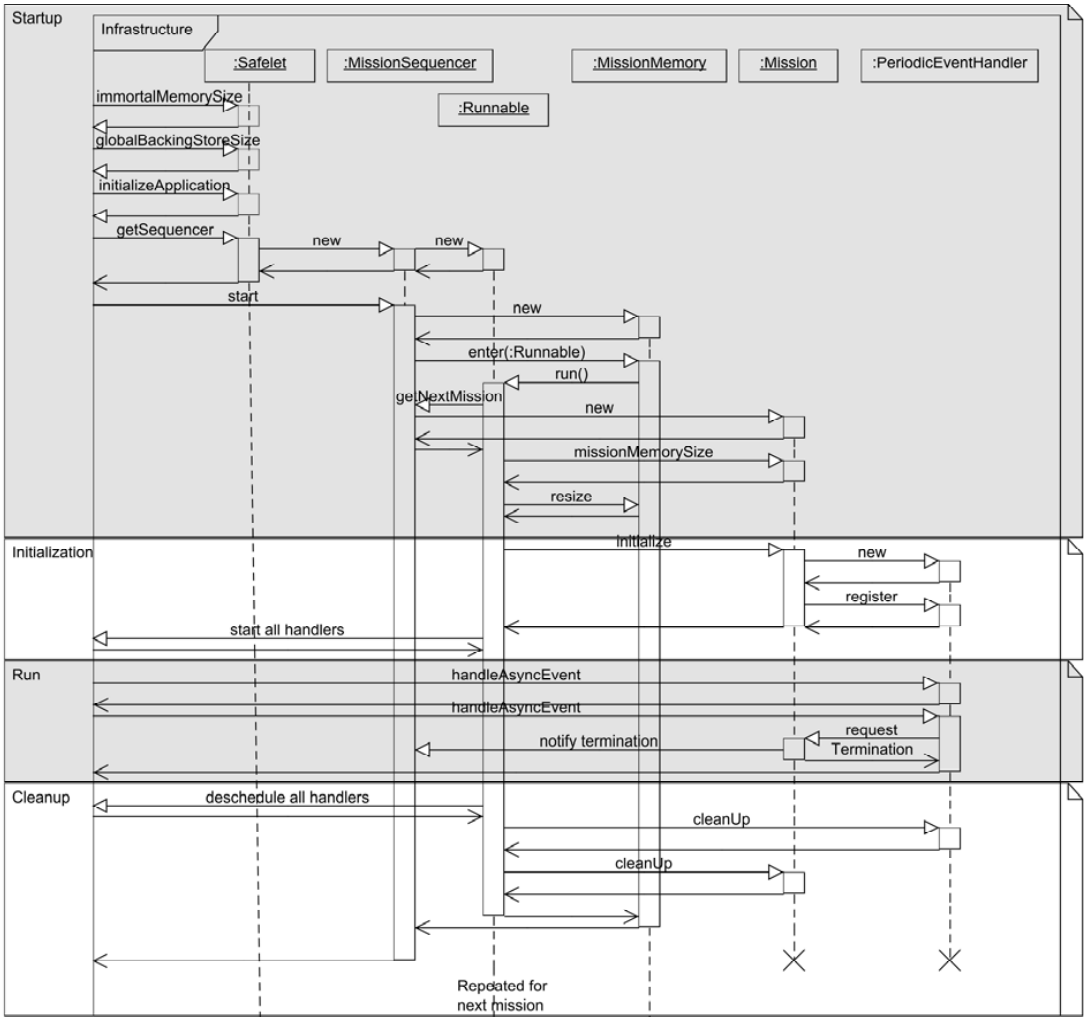

Figure 3.3: Sample Level 1 Lifecycle Sequence Diagram

executing application code before the entire application has been loaded. It also allows application programs to run with unresolved references, as long as the path that makes use of the unresolved reference is never exercised. This capability is useful during prototyping and incremental developmenet, as it allows experimentation with particular designs and planned features even before the complete system has been implemented.

The Java virtual machine (JVM) specification is intentionally vague regarding the time at which classes are loaded, in order to enable deferred class loading as described above. At the same time, the JVM specification is very precise in its characterization of when classes get initialized. In particular, the JVM specification requires that classes be initialized immediately before first use. In compiled implementations of Java, this generally manifests as several extra instructions and a conditional branch in the code that is generated for every access (field or method invocation) to a class.

Deferred class loading and class initialization presents several problems to developers of safety-critical code. Specifically,

- Many safety-critical applications have hard real-time constraints, requiring programmers to accurately derive tight upper bounds on the time required to execute each critical piece of code. When the typical path through a body of code does not involve class loading or class initialization, but every path through the code has to test whether class loading or class initialization is necessary and on rare occassion, the path through this code may require loading and initialization of multiple classes, there is too much variation in the path's execution time.
- The extra code that is generated to force class initialization immediately before first use and code that may be present to enable deferred class loading represents extra code that must be tested at certification time. DO-178C Level A guidelines require "Modified Condition/Decision Coverage" (MC/DC) testing of all conditional branches. It is generally not possible to perform MC/DC testing of each class initialization conditional branch because each class is initialized only once, whereas a typical safety critical application may have thousands of access points to a class, each of which has a conditional branch to test whether this particular access point is required to perform the corresponding class initialization.
- In the presence of circular dependencies between class initialization methods, the uninitialized static data associated with one or more classes may be exposed beyond the boundaries of the class. For example, consider the following simple program, which approximately half of the time initializes A.constant to 8 and B.constant to 11, and the other half of the time initializes A.constant to 11 and B.constant to 3. Depending on which path is taken through the main method's if statement, either class A or class B is seen by the other class in its uninitialized state. Traditional Java issues no error messages or warnings either at compile

or run time. Because of the circular dependencies, this program as written is actually a bit non-sensical.

```
import java.util.Date;
import java.util.Random;

public class Circularity {

  public static class A {
    public final static int constant = B.constant + 8;
  }

  public static class B {
    public final static int constant = A.constant + 3;
  }

  public static void main(String[] args) {
    Date d = new Date();
    Random r = new Random(d.getTime());
    final int a, b;

    if (r.nextFloat() > 0.5) {
      a = A.constant;
      b = B.constant;
    }
    else {
      b = B.constant;
      a = A.constant;
    }
    System.out.println("Constant A equals " + a);
    System.out.println("Constant B equals " + b);
  }
}
```

For these reasons, the SCJ specification requires the absence of circular dependencies among the initialization code that corresponds to each of the classes that comprise an SCJ application. Furthermore, the SCJ specification requires that all classes be loaded and initialized prior to instantiation of the SCJ application's Safelet class.

Note that the requirement to load and initialize all classes prior to the start of an SCJ application is fully compatible with existing Java virtual machines provided that a main Java program includes code that accesses each of the classes that is part of the SCJ application before the main program arranges to instantiate the SCJ application's Safelet object. In the absence of cycles, the underlying JVM implementation will arrange to initialize all classes in a topological sort order according to class initialization dependencies, regardless of the order in which the individual classes are accessed.

If a particular vendor desires to provide enhanced capabilities to support dynamic

class loading, such capabilities are strictly outside the specification for SCJ.

3.6.2 MissionSequencer as a ManagedEventHandler

Mission sequencers appears in two contexts. A MissionSequencer object is included to oversee execution of the SCJ application. And each Level 2 mission may include among its ManagedSchedulables one or more mission sequencers which oversee the execution of inner-nested missions. In both cases, the mission sequencer depends on an associated bound thread to perform certain actions, such as selection of the next mission to run, initialization of that mission before it enters its execution phase, and cleaning up the mission's shared data structures after it finishes its execution phase.

Since mission sequencers play a critical role in all three SCJ levels, it was decided to structure the MissionSequencer type as a subclass of ManagedEventHandler even though it might have been more natural to treat it as a subclass of ManagedThread. This is because SCJ levels zero and one do not support the ManagedThread class.

To enable reliable and consistent operation of the MissionSequencer, it is important that the MissionSequencer constructors allow specification of its corresponding ScopeParameters. In the case that the ScopeParameters specified for a Safelet's outermost MissionSequencer are consistent with the resources already available to the Safelet's initialization thread, it is intended that a compliant SCJ implementation may use the same thread to perform Safelet initialization and mission sequencing.

3.6.3 Sizing of Mission Memories

Multiple perspectives and programming styles were considered in the design of the mission sequencing and mission APIs. Among perspectives was a desire to allow simple programs to be implemented with minimal effort. In contrast, there was a competing desire to enable strong separation of concerns, encapsulation, and abstraction capabilities for the implementation of large and complex safety-critical systems.

In order to support both perspectives, the resulting API allows programmers to choose where Mission objects reside in relation to the corresponding mission memory. For simpler applications, it may be desirable for the Mission object to reside in memory external to the corresponding mission memory area. Note, for example, that the API for the LinearMissionSequencer requires that all of the missions to be executed be passed in as constructor arguments. Thus, these missions must reside in a memory area that is external to the mission memory area that will correspond to the mission itself.

In more complex systems, it may be preferable to allocate the mission object within its own mission memory area. This has the following benefits. First, the mission

is guaranteed to begin executing in its virgin newly constructed state. When programmers allocate missions in outer-nested memory areas, the programmer needs to provide additional code to restore that mission to an appropriate state before the same mission is restarted. Also, programmers must manage the additional complexity that might arise if the same mission is allowed to run simultaneously under the direction of multiple nested mission sequencers. Second, this mission object is allowed to refer directly to the objects that are allocated within mission memory by the mission's initialize method, including the various ManagedSchedulable objects that comprise this mission.

To support encapsulation, it was decided that the required size of mission memory should be represented by an instance method of the corresponding Mission object. To support the programming style in which the Mission is allocated within the mission memory area, the mission memory area is allocated and initialized before the Mission object is allocated. Thus, at the time mission memory is initialized, the infrastructure does not know how big to make the associated backing store. It was therefore decided that immediately before invocation of the MissionSequencer's getNextMission method, the infrastructure will size the mission memory area to include all available backing store memory associated with the MissionSequencer. Upon return from the getNextMission method, the infrastructure invokes the returned Mission object's missionMemorySize method and then truncates the mission memory area's size to the requested size before invoking the mission's initialize method. A consequence of this API design choice is that the implementation of getNextMission may not introduce private memory areas to perform temporary allocations.

3.6.4 Hierachical Decomposition of Memory Resources

One of the design goals of the SCJ specification has been to enable the development of SCJ applications that are not vulnerable to reliability failures due to memory fragmentation. Earlier drafts of this SCJ specification described a hierarchical decomposition of all memory such that all of the memory dedicated to a mission could be divided into smaller portions, each dedicated to the reliable execution of one of the mission's managed schedulable objects. It was then thought that associating three contiguous regions of memory for the dedicated use of each SCJ schedulable object is sufficient to assure reliable operation of the thread. Conceptually, the three regions of memory correspond to reservations for scoped memory area backing stores, the Java stack as required for interpretation of Java bytecodes, and a native stack for implementation of bytecodes and native methods. The earlier draft specification allowed applications to specify the memory requirements for each schedulable object, and described the decomposition of the memory associated with a mission sequencer into independent memory segments to represent the needs of each of the currently running mission's managed schedulables.

However, that earlier API memory design was abandoned in the final draft because of concerns that it would be too difficult to support that API on certain of the platforms being considered to be relevant for execution of SCJ applications. In particular, when running on top of certain real-time operating systems, it is not possible to force a newly spawned thread to use a particular portion of an existing thread's run-time stack as its run-time stack.

In the current specification, the ScopeParameters object associated with every managed schedulable object addresses only the hierarchical decomposition of its backing store memory. The ConfigurationParameters provide additional information. The sizes array provides an opportunity for individual vendors to provide mechanisms that assure the absence of fragmentation of stack memory on particular platforms. For example, a vendor might specify that sizes[0] represents the Java stack memory budget for the newly created thread, which is always to be satisfied by taking a contiguous portion from the Java stack memory budget for the corresponding mission sequencer's thread. Likewise, sizes[1] might be defined to represent the native stack memory budget for the newly created thread, which is always to be satisfied by taking a contiguous portion of the native stack memory budget for the corresponding mission sequencer's thread.

Besides eliminating memory fragmentation risks, a second design goal of the SCJ specification is to eliminate out-of-memory conditions for every scope and to prevent stack overflow conditions for every thread. After reviewing proposals for standardizing solutions to these problems no single approach achieved a sufficient level of consensus to be included in the standard. Thus, the SCJ specification leaves it to individual developers and vendors of SCJ implementations to develop proprietary techniques for analyzing the size requirements of each scope, as well as the cumulative stack memory requirements for each thread.

3.6.5 Some Style Recommendations Regarding Design of Missions

When sharing mission data between the multiple schedulable objects that comprise a particular mission, the programmer must decide between several alternative mechanisms for providing the individual schedulable objects with references to the shared data structures.

- The individual schedulable objects may invoke Mission.getMission, coerce the result to the known Mission subclass, and directly access the fields and methods of this Mission subclass to obtain access to the shared mission data.
- Alternatively, references to the shared data objects may be passed in as arguments to the constructors of each of the relevant managed schedulable objects.

Though the software engineering tradeoffs must be assessed by developers in each specific context, it is generally considered that the latter approach constitutes a better style. There are several reasons for this. First, the flow of information is clearly delineated by the constructor's parameterization. Second, the mission itself is able to more easily restrict access to its shared data by declaring the relevant fields and methods to be private. This makes it easier to enforce sharing information only among components that truly need access to that information. Third, the implementation of individual schedulable objects can be made more independent of the mission within which they run. Since the schedulable object doesn't need to know the type of the mission that hosts its execution, an application component may be more easily reused in different contexts, under the oversight of a different Mission subtype.

3.6.6 Comments on Termination of Missions

With simple missions comprised entirely of AperiodicEventHandler and PeriodicEventHandler schedulable objects, termination of the mission is fairly automatic. When application code invokes the mission's requestTermination method, the infrastructure arranges to disable all further releases of the corresponding event handlers. All of the managed schedulable objects associated with the mission will terminate upon completion of any currently released event handlers.

Termination of Level 2 missions that include the execution of ManagedThread schedulables is a bit more complex. This is because there's no natural stopping point for a running thread. Instead, the thread must stop itself at an appropriate application-specific time. In order to coordinate with running threads, the SCJ API provides the signalTermination method in the ManagedSchedulable interface. Termination protocols can be supported by overriding the signalTermination method in the SCJ classes that implement the ManagedSchedulable interface. The overridden method might invoke, for example, Thread.interrupt if the thread could be blocked in the Object.wait method.

Potentially, every mission includes some application-specific termination code. Thus, it is generally good practice for every application-specific overriding of the signalTermination method to include an invocation of the super class's method.

Finally, with all code that manipulates shared state, if synchronized code is used during the implementation of any application-level termination protocols, issues of potential deadlocks and race conditions need to be considered.

3.6.7 Special Considerations for Level 0 Missions

Within a Level 0 execution environment, periodic event handlers are scheduled by a static cyclic executive. Within this environment, the PeriodicParameters and Priori-

tyParameters arguments to the PeriodicEventHandler constructors are ignored at run time.

It was decided that SCJ would keep the same parameterization of PeriodicEventHandler constructors for all SCJ levels for consistency reasons. The presence of these arguments even in a Level 0 application helps document the intent of the code. Presumably, the static cyclic schedule that governs execution of periodic event handlers is consistent with the behavior of a dynamic scheduler based on the values of the PeriodicParameters and PriorityParameters arguments.

Every CyclicExecutive object is required to provide an implementation of the CyclicExecutive.getSchedule method. This method returns the CyclicSchedule object which represents the static cyclic schedule that governs execution of the mission's PeriodicEventHandler activities. The SCJ specification does not concern itself with how this schedule is generated, though it expects that vendors who provide compliant implementations of the SCJ specification and third party tool vendors are likely to provide tools to automate the creation of these schedules.

One benefit of using the same constructor parameterization of PeriodicEventHandler objects in all levels is that a Level 0 mission can run within a Level 1 or Level 2 runtime environment. If a CyclicExecutive mission is selected for execution by a Level 1 or Level 2 mission sequencer, the periodic event handlers will be scheduled dynamically in that context, based on the values of the constructor's PeriodicParameters and PriorityParameters arguments, and the mission's CyclicExecutive.getSchedule method will not be invoked by infrastructure.

Given this generality, which allows CyclicExecutive missions to run within Level 1 and Level 2 execution environments, it is evident that only Level 0 missions that use Java synchronized methods to access all data and other resources shared among multiple periodic event handlers will execute correctly in a Level 1 or Level 2 execution environment,

3.6.8 Implementation of MissionSequencers and Missions

From the application programmer's perspective, ManagedSchedulable objects are nested within the Mission object with which they are associated, and each Mission is nested within a MissionSequencer object's context. This hierarchy represents a logical decomposition that matches recommended software engineering practices to break large and complex problems into smaller parts that can be independently managed more easily than tackling the entire system as a monolithic body of code.

The implementation of this abstraction is made somewhat more complex by the scoped memory rules of the RTSJ. In particular, a mission sequencer is expected to keep track of the mission that is running within it, because an invocation of the se-

quencer's signalTermination must result in an invocation of the currently running mission's requestTermination method. Likewise, a mission is expected to keep track of all its associated ManagedSchedulable objects because an invocation of its requestTermination method is expected to send shut-down requests to each of the corresponding threads and then wait for each of them to terminate.

Since missions may reside in scopes that nest internal to the mission that holds a mission sequencer, and since each managed schedulable object may reside in a scope that nests within the scope that holds the corresponding mission object, it is not generally possible for mission sequencer objects to refer directly to the mission object that represents the currently running mission. For the same reasons, it is not possible for missions to, in general, hold direct references to the managed schedulable objects associated with the mission.

An implementation technique that is used in the official SCJ reference implementation is to use the MissionSequencer thread's local variables to hold references to inner-nested objects. This thread can, for example, store a reference to the currently running mission in a local variable and can store each of the mission's associated managed schedulable objects within a local array. The thread then blocks itself on a condition associated with this mission and its sequencer. When the condition is notified, the thread becomes unblocked so that it can perform services on behalf of the thread that was responsible for notification. Notification would occur, for example, if the MissionSequencer is part of a nested mission and that mission has been requested to terminate.

3.6.9 Example of a Static Level 0 Application

This section provides an example of a simple Level 0 application. Note that the SimpleCyclicExecutive class both extends CyclicExecutive and implements Safelet<CyclicExecutive>. The application begins with instantiation of this class.

3.6.10 SimpleCyclicExecutive.java

```
package samples.staticlevel0;

import javax.realtime.PriorityParameters;
import javax.realtime.RelativeTime;
import javax.realtime.memory.ScopeParameters;

import javax.safetycritical.CyclicExecutive;
import javax.safetycritical.CyclicSchedule;
import javax.safetycritical.LinearMissionSequencer;
import javax.safetycritical.MissionSequencer;
```

```
import javax.safetycritical.PeriodicEventHandler;
import javax.safetycritical.Safelet;

import javax.safetycritical.annotate.SCJAllowed;
import static javax.safetycritical.annotate.Level.SUPPORT;

@SCJAllowed(members=true)
public class SimpleCyclicExecutive
  extends CyclicExecutive
  implements Safelet<CyclicExecutive>
{
  final int MISSION_MEMORY_SIZE = 10000;
  final int IMMORTAL_MEMORY_SIZE = 10000;
  final int SEQUENCER_PRIORITY = 10;

  public void initializeApplication() {
    ;
  }

  public long missionMemorySize()
  {
    return MISSION_MEMORY_SIZE;
  }

  public void initialize() {
    (new MyPEH("A",new RelativeTime(0,0),new RelativeTime(500,0))).register();
    (new MyPEH("B",new RelativeTime(0,0),new RelativeTime(1000,0))).register();
    (new MyPEH("C",new RelativeTime(0,0),new RelativeTime(500,0))).register();
  }

  @SCJAllowed(SUPPORT)
  public CyclicSchedule
  getSchedule(PeriodicEventHandler[] pehs) {
    return VendorCyclicSchedule.generate(pehs, this);
  }

  // Safelet methods

  @SCJAllowed(SUPPORT)
  public MissionSequencer<CyclicExecutive> getSequencer()
  {
    // The returned LinearMissionSequencer is allocated in ImmortalMemory
    return new LinearMissionSequencer<CyclicExecutive>(
      new PriorityParameters(SEQUENCER_PRIORITY),
      new ScopeParameters(10000, 100,0,0),
      this);
  }

  public long immortalMemorySize()
  {
```

```
    return IMMORTAL_MEMORY_SIZE;
  }
}
```

3.6.11 MyPEH.java

```
package samples.staticlevel0;

import javax.realtime.PeriodicParameters;
import javax.realtime.PriorityParameters;
import javax.realtime.memory.ScopeParameters;
import javax.realtime.RelativeTime;
import javax.safetycritical.PeriodicEventHandler;
import javax.safetycritical.annotate.SCJAllowed;

import static javax.safetycritical.annotate.Level.SUPPORT;

@SCJAllowed(members=true)
public class MyPEH extends PeriodicEventHandler {

  static final int priority = 13, mSize = 10000;
  int eventCounter;
  String my_name;

  public MyPEH(String nm, RelativeTime start, RelativeTime period) {
    super(new PriorityParameters(priority),
          new PeriodicParameters(start, period),
          new ScopeParameters(10000, 0,100,0), null);
    my_name = nm;
  }

  @SCJAllowed(SUPPORT)
  public void handleAsyncEvent() {
    ++eventCounter;
  }
}
```

3.6.12 VendorCyclicSchedule.java

```
package samples.staticlevel0;

import javax.realtime.RelativeTime;

import javax.safetycritical.CyclicExecutive;
import javax.safetycritical.CyclicSchedule;
import javax.safetycritical.PeriodicEventHandler;
import javax.safetycritical.annotate.SCJAllowed;


@SCJAllowed(members=true)
```

```
class VendorCyclicSchedule {

  static CyclicExecutive cache_key;
  static CyclicSchedule cache_schedule;

  private PeriodicEventHandler[] peh;

  /*
   * Instantiate a vendor-specific cyclic schedule and return it.
   * Note that in normal usage, this executes in mission memory.
   *
   * This sample implementation of a CyclicSchedule generator presents
   * the code that might be automatically generated by a vendor-specific
   * tool.
   *
   * In this example, the generated schedule is for an application
   * that has three asynchronous event handlers to be dispatched.
   * There are two frames for the application. The first frame has an
   * offset of 0 from the start time and runs PEH A followed by PEH B,
   * in order. The second frame has an offset of 500ms from the start
   * time and runs PEH A followed by PEH C, in order.
   */
  static CyclicSchedule generate(PeriodicEventHandler[] peh,
                                 CyclicExecutive m) {
    if (m == cache_key)
      return cache_schedule;
    else {
      //
      // For simplicity of presentation, the following five
      // allocations are taken from mission memory. A more frugal
      // implementation would allocate these objects in private memory.
      //
      CyclicSchedule.Frame frames[] = new CyclicSchedule.Frame[2];
      PeriodicEventHandler frame1_handlers[] = new PeriodicEventHandler[3];
      PeriodicEventHandler frame2_handlers[] = new PeriodicEventHandler[2];
      RelativeTime frame1_duration = new RelativeTime(500, 0);
      RelativeTime frame2_duration = new RelativeTime(500, 0);

      frame1_handlers[0] = peh[0]; // A
      frame1_handlers[1] = peh[2]; // C scheduled before B due to RMA
      frame1_handlers[2] = peh[1]; // B

      frame2_handlers[0] = peh[0]; // A
      frame2_handlers[1] = peh[2]; // C

      frames[0] = new CyclicSchedule.Frame(frame1_duration, frame1_handlers);
      frames[1] = new CyclicSchedule.Frame(frame2_duration, frame2_handlers);

      cache_schedule = new CyclicSchedule(frames);
      cache_key = m;
```

```
        return cache_schedule;
      }
    }
  }
```

3.6.13 Example of a Dynamic Level 0 Application

The example above allocates the SimpleCyclicExecutive application in ImmortalMemory. The example described in this section allocates the same SimpleCyclicExecutive object in mission memory. For illustrative purposes, this example repeatedly executes the SimpleCyclicExecutive mission. Each time the SimpleCyclicExecutive mission terminates, the mission memory area is exited and all of the objects allocated within it, including the SimpleCylicExecutive object are reclaimed. In this example, a new SimpleCyclicExecutive object is allocated for each new execution by the getNextMission method.

For simplicity of presentation, we are reusing the existing SimpleCyclicExecutive object even though it is more general than we need for this particular example. In this example, we ignore the fact that SimpleCyclicExecutive implements the Safelet interface.

The implementation of LinearMissionSequencer requires that the sequenced missions reside external to the mission memory area. In order to arrange for Mission objects to be newly allocated within the mission memory area immediately before each mission execution, it is necessary for the developer to implement a subclass of MissionSequencer.

3.6.14 MyLevel0App.java

```
package samples.dynamiclevel0;

import javax.realtime.PriorityParameters;

import javax.safetycritical.CyclicExecutive;
import javax.safetycritical.MissionSequencer;
import javax.safetycritical.Safelet;
import javax.realtime.memory.ScopeParameters;

import javax.safetycritical.annotate.SCJAllowed;
import javax.safetycritical.annotate.SCJPhase;

import static javax.safetycritical.annotate.Level.LEVEL_0;
import static javax.safetycritical.annotate.Level.SUPPORT;
import static javax.safetycritical.annotate.Phase.INITIALIZATION;
```

```
@SCJAllowed
class MyLevel0App implements Safelet<CyclicExecutive> {

  @SCJAllowed(LEVEL_0)
  public MyLevel0App() {
  }

  @SCJAllowed(LEVEL_0)
  public void intializeApplication() {
    ; // do nothing
  }

  @SCJAllowed(SUPPORT)
  @SCJPhase({INITIALIZATION})
  public MissionSequencer<CyclicExecutive> getSequencer() {
    PriorityParameters p = new PriorityParameters(18);
    ScopeParameters s = new ScopeParameters(100000, null, 80, 512);
    return new MyLevel0Sequencer(p, s);
  }

  public long immortalMemorySize() {
    return 10000l;
  }
}
```

3.6.15 MyLevel0Sequencer.java

```
package samples.dynamiclevel0;

import javax.realtime.PriorityParameters;
import javax.realtime.memory.ScopeParameters;

import javax.safetycritical.CyclicExecutive;
import javax.safetycritical.MissionSequencer;

import javax.safetycritical.annotate.SCJAllowed;
import javax.safetycritical.annotate.SCJPhase;

import static javax.safetycritical.annotate.Level.LEVEL_0;
import static javax.safetycritical.annotate.Level.SUPPORT;
import static javax.safetycritical.annotate.Phase.INITIALIZATION;

import samples.staticlevel0.SimpleCyclicExecutive;

@SCJAllowed
class MyLevel0Sequencer extends MissionSequencer<CyclicExecutive> {

  @SCJAllowed(LEVEL_0)
  public MyLevel0Sequencer(PriorityParameters p, ScopeParameters s) {
    super(p, s);
```

```
  }

  @SCJAllowed(SUPPORT)
  @SCJPhase({INITIALIZATION})
  protected CyclicExecutive getNextMission() {
    return new SimpleCyclicExecutive();
  }
}
```

3.6.16 Example of a Level 1 Application

The simple Level 1 application presented in this section reuses the MyPEH implementation from the static Level 0 application. As with that example, note that MyLevel1App both extends Mission and implements Safelet<Mission>.

3.6.17 MyLevel1App.java

```
package samples.level1;

import javax.realtime.PriorityParameters;
import javax.realtime.RelativeTime;
import javax.safetycritical.LinearMissionSequencer;
import javax.safetycritical.Mission;
import javax.safetycritical.MissionSequencer;
import javax.safetycritical.Safelet;
import javax.realtime.memory.ScopeParameters;
import javax.safetycritical.annotate.SCJAllowed;

import static javax.safetycritical.annotate.Level.SUPPORT;

import samples.staticlevel0.MyPEH;

@SCJAllowed(members=true)
public class MyLevel1App
  extends Mission
  implements Safelet<Mission>
{
  final int MISSION_MEMORY_SIZE = 10000;
  final int SEQUENCER_PRIORITY = 10;

  public void intializeApplication()
  {
    ; // do nothing
  }

  public long missionMemorySize()
  {
    return MISSION_MEMORY_SIZE;
```

```
  }

  public void initialize() {
    // Note that MyPEH, imported from samples.staticlevel0,
    // generalizes to execution in a level−1 environment. When
    // running in level−0, the start and period arguments were
    // ignored because the level−0 dispatcher simply runs the computed
    // static cyclic schedule.
    (new MyPEH("A",new RelativeTime(0,0),new RelativeTime(500,0))).register();
    (new MyPEH("B",new RelativeTime(0,0),new RelativeTime(1000,0))).register();
    (new MyPEH("C",new RelativeTime(0,0),new RelativeTime(500,0))).register();
  }

  // Safelet methods

  public MissionSequencer<Mission> getSequencer()
  {
    // The returned LinearMissionSequencer is allocated in ImmortalMemory
    return new LinearMissionSequencer<Mission>(
      new PriorityParameters(SEQUENCER_PRIORITY),
      new ScopeParameters(10000, null),
      this);
  }

  public long immortalMemorySize() {
    return 10000;
  }
}
```

3.6.18 Example of a Level 2 Application

The following code illustrates how a simple Level 2 application could be written with nested missions. Figure 3.4 illustrates the sequence of activities that comprise execution of this Level 2 example.

3.6.19 MyLevel2App.java

```
package samples.level2;

import javax.realtime.PriorityParameters;
import javax.realtime.PriorityScheduler;
import javax.realtime.memory.ScopeParameters;

import javax.safetycritical.Mission;
import javax.safetycritical.MissionSequencer;
import javax.safetycritical.Safelet;
import javax.safetycritical.annotate.SCJAllowed;
```

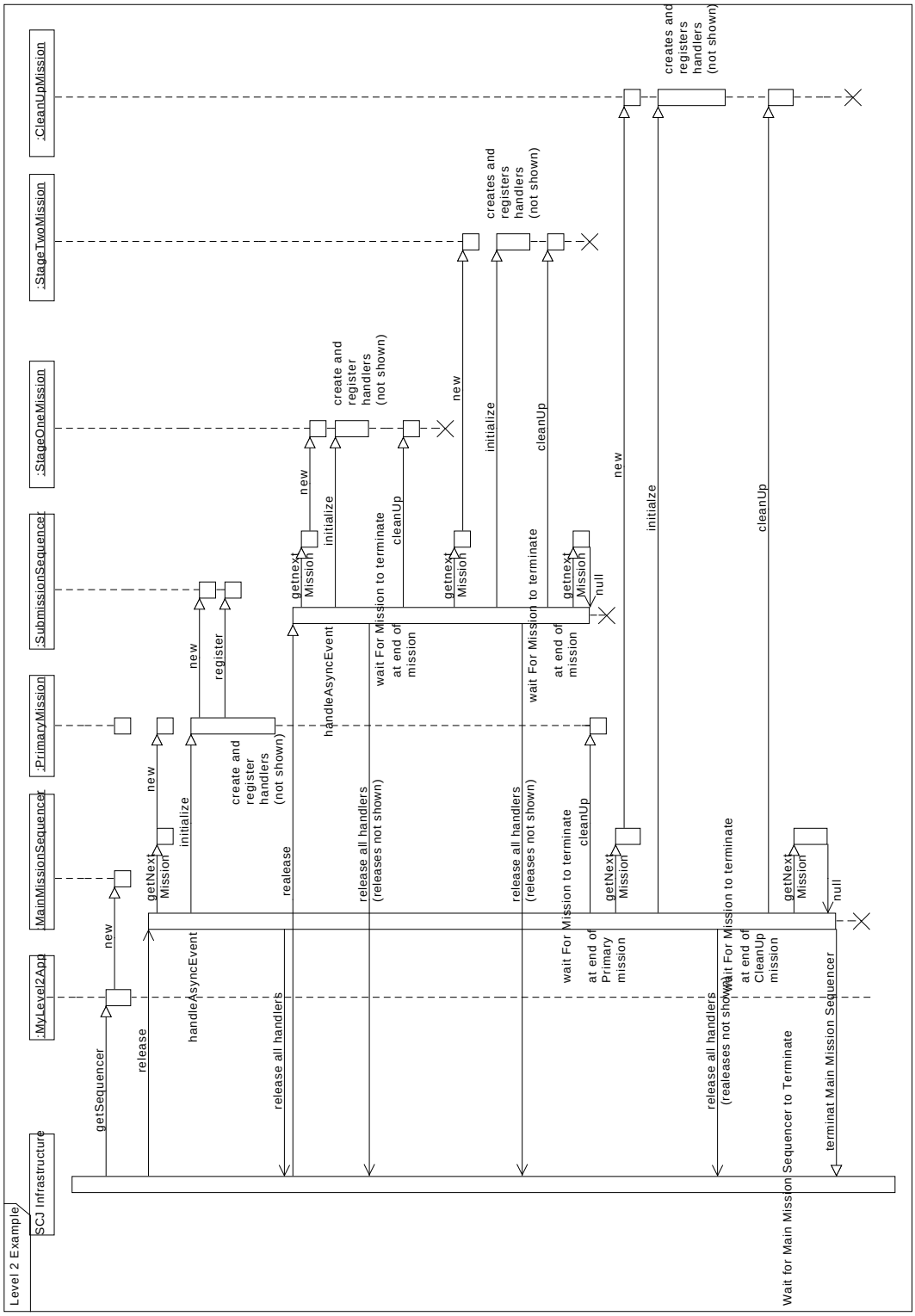

Figure 3.4: UML sequence diagram for Level 2 example

```
import static javax.safetycritical.annotate.Level.LEVEL_2;


@SCJAllowed(members=true, value=LEVEL_2)
public class MyLevel2App implements Safelet<Mission> {

  static final private int PRIORITY =
    PriorityScheduler.instance().getNormPriority();

  public void initializeApplication() {
    ; // do nothing
  }

  public MissionSequencer<Mission> getSequencer() {
    ScopeParameters sp =
      new ScopeParameters(100000L, null);
    return new MainMissionSequencer(new PriorityParameters(PRIORITY), sp);
  }

  public long immortalMemorySize() {
    return 10000;
  }

}
```

3.6.20 MainMissionSequencer.java

```
package samples.level2;

import javax.realtime.PriorityParameters;
import javax.realtime.memory.ScopeParameters;

import javax.safetycritical.Mission;
import javax.safetycritical.MissionSequencer;

import javax.safetycritical.annotate.SCJAllowed;

import static javax.safetycritical.annotate.Level.LEVEL_2;
import static javax.safetycritical.annotate.Level.SUPPORT;

@SCJAllowed(members=true, value=LEVEL_2)
public class MainMissionSequencer extends MissionSequencer<Mission> {

  private boolean initialized, finalized;

  MainMissionSequencer(PriorityParameters priorityParameters,
                       ScopeParameters storageParameters) {
    super(priorityParameters, storageParameters);
    initialized = finalized = false;
  }
```

```
  @SCJAllowed(SUPPORT)
  protected Mission getNextMission() {
    if (finalized)
      return null;
    else if (initialized) {
      finalized = true;
      return new CleanupMission();
    }
    else {
      initialized = true;
      return new PrimaryMission();
    }
  }
}
```

3.6.21 PrimaryMission.java

```
package samples.level2;

import javax.realtime.PriorityParameters;
import javax.realtime.PriorityScheduler;
import javax.realtime.RelativeTime;
import javax.realtime.memory.ScopeParameters;

import javax.safetycritical.Mission;

import javax.safetycritical.annotate.SCJAllowed;

import static javax.safetycritical.annotate.Level.LEVEL_2;

@SCJAllowed(members=true, value=LEVEL_2)
public class PrimaryMission extends Mission {
  final private int MISSION_MEMORY_SIZE = 10000;

  static final private int PRIORITY =
    PriorityScheduler.instance().getNormPriority();

  public long missionMemorySize() {
    return MISSION_MEMORY_SIZE;
  }

  public void initialize() {
    PriorityParameters pp = new PriorityParameters(PRIORITY);
    ScopeParameters sp =
      new ScopeParameters(100000L, null);
    SubMissionSequencer sms = new SubMissionSequencer(pp, sp);
    sms.register();
    (new MyPeriodicEventHandler("AEH_A", new RelativeTime(0, 0),
                                new RelativeTime(500, 0))).register();
```

```
    (new MyPeriodicEventHandler("AEH B", new RelativeTime(0, 0),
                                        new RelativeTime(1000, 0))).register();
    (new MyPeriodicEventHandler("AEH C", new RelativeTime(500, 0),
                                      new RelativeTime(500, 0))).register();
  }
}
```

3.6.22 CleanupMission.java

```
package samples.level2;

import javax.realtime.PriorityParameters;
import javax.realtime.PriorityScheduler;

import javax.safetycritical.Mission;
import javax.realtime.memory.ScopeParameters;

import javax.safetycritical.annotate.SCJAllowed;

import static javax.safetycritical.annotate.Level.LEVEL_2;

@SCJAllowed(members=true, value=LEVEL_2)
public class CleanupMission extends Mission {
  static final private int MISSION_MEMORY_SIZE = 10000;
  static final private int PRIORITY =
    PriorityScheduler.instance().getNormPriority();

  public long missionMemorySize() {
    return MISSION_MEMORY_SIZE;
  }

  public void initialize() {
    PriorityParameters pp = new PriorityParameters(PRIORITY);
    ScopeParameters sp =
      new ScopeParameters(100000L, null);
    MyCleanupThread t = new MyCleanupThread(pp, sp);
  }
}
```

3.6.23 SubMissionSequencer.java

```
package samples.level2;

import javax.realtime.PriorityParameters;
import javax.realtime.memory.ScopeParameters;

import javax.safetycritical.Mission;
import javax.safetycritical.MissionSequencer;

import javax.safetycritical.annotate.SCJAllowed;
```

```
import static javax.safetycritical.annotate.Level.LEVEL_2;

@SCJAllowed(members=true, value=LEVEL_2)
public class SubMissionSequencer extends MissionSequencer<Mission> {
  private boolean initialized, finalized;

  SubMissionSequencer(PriorityParameters priorityParameters,
                      ScopeParameters storageParameters) {
    super(priorityParameters, storageParameters);
    initialized = finalized = false;
  }

  protected Mission getNextMission() {
    if (finalized)
      return null;
    else if (initialized) {
      finalized = true;
      return new StageTwoMission();
    }
    else {
      initialized = true;
      return new StageOneMission();
    }
  }
}
```

3.6.24 StageOneMission.java

```
package samples.level2;

import javax.realtime.RelativeTime;

import javax.safetycritical.Mission;

import javax.safetycritical.annotate.SCJAllowed;


import static javax.safetycritical.annotate.Level.LEVEL_2;


@SCJAllowed(members=true, value=LEVEL_2)
public class StageOneMission extends Mission {
  private static final int MISSION_MEMORY_SIZE = 10000;

  public long missionMemorySize() {
    return MISSION_MEMORY_SIZE;
  }

  public void initialize() {
```

```
    (new MyPeriodicEventHandler("stage1.eh1",
                                new RelativeTime(0, 0),
                                new RelativeTime(1000, 0))).register();
  }
}
```

3.6.25 StageTwoMission.java

```
package samples.level2;

import javax.realtime.RelativeTime;

import javax.safetycritical.Mission;

import javax.safetycritical.annotate.SCJAllowed;

import static javax.safetycritical.annotate.Level.LEVEL_2;


@SCJAllowed(members=true, value=LEVEL_2)
public class StageTwoMission extends Mission {
  private static final int MISSION_MEMORY_SIZE = 10000;

  public long missionMemorySize() {
    return MISSION_MEMORY_SIZE;
  }

  public void initialize() {
    (new MyPeriodicEventHandler("stage2.eh1",
                                new RelativeTime(0, 0),
                                new RelativeTime(500, 0))).register();
  }
}
```

3.6.26 MyPeriodicEventHandler.java

```
package samples.level2;

import javax.realtime.PeriodicParameters;
import javax.realtime.PriorityParameters;
import javax.realtime.RelativeTime;
import javax.realtime.memory.ScopeParameters;

import javax.safetycritical.PeriodicEventHandler;

import javax.safetycritical.annotate.SCJAllowed;

import static javax.safetycritical.annotate.Level.LEVEL_2;

@SCJAllowed(members=true, value=LEVEL_2)
```

```
class MyPeriodicEventHandler extends PeriodicEventHandler {
  private static final int _priority = 17;
  private static final int _memSize = 5000;
  private int _eventCounter;

  public MyPeriodicEventHandler(String aehName,
                                RelativeTime startTime,
                                RelativeTime period) {
    super(new PriorityParameters(_priority),
          new PeriodicParameters(startTime, period),
          new ScopeParameters(10000, null),
          0, aehName);
  }

  public void handleAsyncEvent() {
    ++_eventCounter;
  }

  public void cleanUp() {}
}
```

3.6.27 MyCleanupThread.java

```
package samples.level2;

import javax.realtime.PriorityParameters;
import javax.realtime.memory.ScopeParameters;

import javax.safetycritical.ManagedThread;

import javax.safetycritical.annotate.SCJAllowed;

import static javax.safetycritical.annotate.Level.LEVEL_2;
import static javax.safetycritical.annotate.Level.SUPPORT;

@SCJAllowed(members=true, value=LEVEL_2)
class MyCleanupThread extends ManagedThread {

  public MyCleanupThread(PriorityParameters pp, ScopeParameters sp) {
    super(pp, sp, 0);
  }

  @SCJAllowed(SUPPORT)
  public void run() {
    cleanupThis();
    cleanupThat();
  }

  @SCJAllowed
  void cleanupThis() {
```

```
    // code not shown
  }

  @SCJAllowed
  void cleanupThat() {
    // code not shown
  }
}
```

Chapter 4

Concurrency and Scheduling Models

This chapter describes how concurrency is provided and managed for SCJ applications. The Overview and Rationale sections are not normative but are provided to improve understanding of the normative sections. All of the other sections of this chapter are normative.

4.1 Overview

Many safety-critical systems are small and sequential, relying on cyclic executive scheduling to manually interleave the execution of all activities within their time constraints, but without introducing concurrency. For larger and more complex safety-critical systems, there has been a gradual migration to programming models that support simple concurrent activities (including threads, tasks, event handlers etc) that share an address space with each other.

For this reason, the SCJ defines three compliance levels that reflect various levels of complexity that can occur in a safety-critical application. As a consequence, the SCJ support provided for concurrent programming and scheduling is more comprehensive at the higher compliance levels. This Chapter presents the facilities defined by SCJ at each of its compliance levels.

In general, there are two models for creating concurrent programs. The first is a thread-based model in which each concurrent entity is represented by a thread of control. The second is an event-based model, where an event handler executes in direct response to each event. The RTSJ, upon which this SCJ specification is based, supports a rich concurrency model allowing real-time threads (both heap-using and no-heap) and asynchronous events (also both heap-using and no-heap, and their event handlers). The SCJ concurrency model simplifies this and relies, almost exclusively,

on asynchronous event handling. The reasons for this are pragmatic rather than dogmatic:

1. Real-time threads do not have an easily identifiable section of code that represents an individual release, also called a *job*, in the real-time scheduling community's terminology. In the thread context, a job is the area of code inside a loop that is delimited by a call to the waitForNextPeriod or waitForNextRelease methods. In contrast, an event handler has the handleAsyncEvent method which exactly represents the notion of a job. Hence the creation of static analysis tools to support safety-critical program development is facilitated.

2. As described in the RTSJ, an asynchronous event handler must execute under control of a thread. The RTSJ permits asynchronous event handlers to be either *unbound*, which means that the asynchronous event handler need only to be bound to a thread before it executes, or *bound*, which means that the asynchronous event handler is permanently bound to a thread when it is created. In terms of execution, a bound asynchronous event handler is equivalent to a real-time thread in functionality and its impact on scheduling. Hence little is lost by using bound asynchronous event handlers instead of real-time threads. Using bound handlers (rather than non-bound handlers) removes any additional latency due to thread binding when a handler is released. They are, therefore, more predictable.

Therefore, the SCJ permits applications to execute only bound asynchronous event handlers at Level 0 and Level 1. At Level 2, both bound asynchronous event handlers and a restricted form of no-heap real-time threads are supported. While this specification uses the term *schedulable object* to refer to code in all levels, in Level 1 and Level 2 implementations, SCJ uses the term *schedulable object* to refer to code that is subject to execution by a preemptible scheduler; hence in these implementations, it refers exclusively to either an object derived from a bound asynchronous event handler or a no-heap real-time thread (called a ManagedThread in the SCJ). In Level 0 implementations, the term *schedulable object* simply refers to a PeriodicEventHandler.

An SCJ asynchronous event handler executes in response to invocation requests (known as *release requests* or *release events*), with the resulting execution of the associated logic referred to as a *release* (or a *job*). Release requests are usually categorized as follows:[1]

- periodic—usually time-triggered,
- sporadic—usually event-triggered, or

[1] Please refer to the RTSJ specification [3] for a more rigorous definition.

- aperiodic— event-triggered or time-triggered.

The SCJ supports communication between SCJ schedulable objects using shared variables and other resources and therefore requires support for synchronization and priority inversion management protocols. On multiprocessor platforms[2], it is assumed that all processors can access all shared data and shared resources, although not necessarily with uniform access times.

SCJ specifies a set of constraints placed on the RTSJ concurrency and scheduling models. SCJ supports this constrained model by defining a new set of classes, all of which are implementable using the concurrency constructs defined by the RTSJ. SCJ requires implementations to support priority ceiling emulation (PCE). It should be noted that this is a departure from the RTSJ standard, as in the RTSJ, priority inheritance is the default priority inversion management protocol.

Scheduling in SCJ is performed in the context of a *scheduling allocation domain*. A scheduling allocation domain of any schedulable object consists of the set of processors on which that schedulable object may be executed. Each schedulable object can be scheduled for execution in only one scheduling allocation domain. At Level 0, only one allocation domain is supported for all schedulable objects; this allocation domain consists of one processor. At Level 1, multiple allocation domains may be supported, but each domain must consist of a single processor. Hence, from a scheduling perspective, a Level 1 system is a fully partitioned system. At Level 2, scheduling domains may consists of one or more processors. By default, all schedulable objects are globally scheduled within an allocation domain. However, a schedulable object can also be constrained to be executed on a single processor in a scheduling allocation domain. Scheduling allocation domains are implemented in terms of affinity sets as defined in the RTSJ. A processor shall not be a member of more than one scheduling allocation domain.

SCJ further extends the RTSJ to support the following:

- Missions – all schedulable objects execute in the context of a mission (see Chapter 2).

4.2 Semantics and Requirements

The SCJ concurrency model is designed to facilitate schedulability analysis techniques that are acceptable to certification authorities, and to aid the construction and

[2]The term *processor* is used in this specification to indicate a Central Processing Unit (CPU) that is capable of physically executing a single thread of control at any point in time. Hence, multicore platforms constitute multiprocessors; platforms that support hyperthreading also constitute multiprocessors. It is assumed that all processors are capable of executing all schedulable objects in whose affinity sets they appear.

deployment of small and efficient Java runtime systems. SCJ also supports cyclic scheduling to provide a familiar execution model for developers of traditional safety-critical systems. This allows them to use Java as well as to facilitate a later migration from traditional architectures to more robust concurrent architectures.

The following requirements apply across all conformance levels.

- The number of processors allocated to the Java platform shall be immutable.
- The number of scheduling allocation domains shall be fixed.
- Only no-heap and non-daemon RTSJ schedulable objects shall be supported (e.g., Java threads are not supported).
- All schedulable objects shall have periodic or aperiodic release parameters – schedulable objects with sporadic release parameters are not supported. Schedulable objects without release parameters are considered to be aperiodic. There is no support for CPU-time monitoring and processing group parameters.
- The default ceiling for locks used by the application and the infrastructure shall be javax.realtime.FirstInFirstOutScheduler.instance().getMaxPriority() (that is, the maximum value for local ceilings – see Section 4.7.5).
- Each schedulable object shall be managed by its enclosing mission.
- The infrastructure shall not synchronize on any instances of classes that are part of the public API.

The following lists the main requirements on application designers.

- Shared objects are represented by classes with synchronized methods. No use of the synchronized statement is allowed. Alternatively, the sharing of variables of primitive data types may use the volatile modifier.
- Use of the Object.wait, Object.notify, and Object.notifyAll methods in Level 2 code shall be invoked only on this.
- Nested calls from one synchronized method to another are allowed. The ceiling priority associated with a nested synchronized method call shall be greater than or equal to the ceiling priority associated with the outer call.
- At all levels, synchronized code shall not self-suspend while holding its monitor lock (for example as a result of an I/O request or the sleep method call). An IllegalMonitorStateException shall be thrown if this constraint is violated and detected by the implementation. Requesting a lock (via a synchronized method) is not considered to be self-suspension.
- Each miss handler that is associated with an application event handler shall extend only AperiodicEventHandler, it shall not extend any other SCJ event handler.

Unless indicated otherwise, the classes defined in this section are thread safe

4.3 Level Considerations

Specific semantics apply at each of the different compliance levels.

4.3.1 Level 0

The following requirements are placed on Level 0 compliance.

- The number of processors allocated to a Level 0 application shall be one.
- Only periodic bound asynchronous event handlers (i.e., PeriodicEventHandler) shall be supported.
- Calls to the Object.wait, Object.notify and Object.notifyAll methods are not allowed.
- Scheduling shall be based on the cyclic executive scheduling approach. Execution of the PeriodicEventHandlers shall be performed as if the infrastructure has provided only a single thread of control that is used for all PeriodicEventHandlers. The PeriodicEventHandlers shall be executed non preemptively. A table-driven approach is acceptable, with the schedule being computed statically off-line in an implementation-defined manner prior to executing the mission.
- An implementation is not required to perform locking for synchronized methods. However, it is strongly recommended that applications use synchronized methods or the volatile modifier to support portability of code between levels so the application can be successfully executed on a Level 1 or a Level 2 implementation.
- There shall be no deadline miss detection facility.

4.3.2 Level 1

The following requirements are placed on Level 1 compliance. Unless explicitly stated, these are in addition to Level 0 requirements.

- Aperiodic and one-shot asynchronous event handlers shall be supported.
- Each eventHandler shall be permanently bound to a thread of control not bound to any other event handler. Further, each such thread of control shall be bound to only a single event handler handler.
- The number of predefined scheduling allocation domains (each represented by an affinity set) shall be equal to the number of processors available to the JVM, each of which contains only a single processor. No dynamic creation of affinity sets is allowed.

- Communication between event handlers running on different processors shall be supported.
- Calls to the Object.wait, Object.notify and Object.notifyAll methods are not allowed.
- The scheduling approach shall be full preemptive priority-based scheduling with at least 28 (software and hardware) priorities, with priority ceiling emulation. A portable application should set priorities using javax.realtime.PriorityScheduler.instance().getMinPriority() and javax.realtime.PriorityScheduler.instance().getMaxPriority() to determine the smallest and largest priorities supported by an SCJ infrastructure. Increasing numeric priority values shall reflect higher priorities. If application portability is a primary concern, the application should use no more than 28 priorities. There shall be no support in SCJ for changing application base priorities.
- The releases of a PeriodicEventHandler shall be triggered using absolute time values.
- The releases of a OneShotEventHandler shall be triggered using absolute or relative time values.
- Deadline miss detection shall be supported. The deadline miss shall be released no earlier than the deadline of the associated event handler whose deadline is being monitored. The deadline miss handler executes at its own priority which shall determine whether it preempts the event handler with which it is associated. An implementation is required to document the time granularity at which missed deadlines are detected (see Section 4.8.5).
- A preempted schedulable object shall be executed as if it were placed at the front of the ready queue for its active priority level. This is a recommendation in the RTSJ but is a requirement for SCJ.

4.3.3 Level 2

The following requirements are placed on Level 2 compliance. Unless explicitly stated, these are in addition to Level 1 requirements.

- No-heap real-time threads shall be supported but shall be managed (the ManagedThread class).
- There shall be a fixed number of implementation-defined scheduling allocation domains (each represented by an affinity set). Each affinity set may contain one or more processors. However, no processor shall appear in more than one domain.
- Dynamic creation of affinity sets is permitted during the mission initialization phase, but each affinity set shall only contain a single processor. The processor identified in the affinity set shall be a member of one of the predefined scheduling allocation domains.

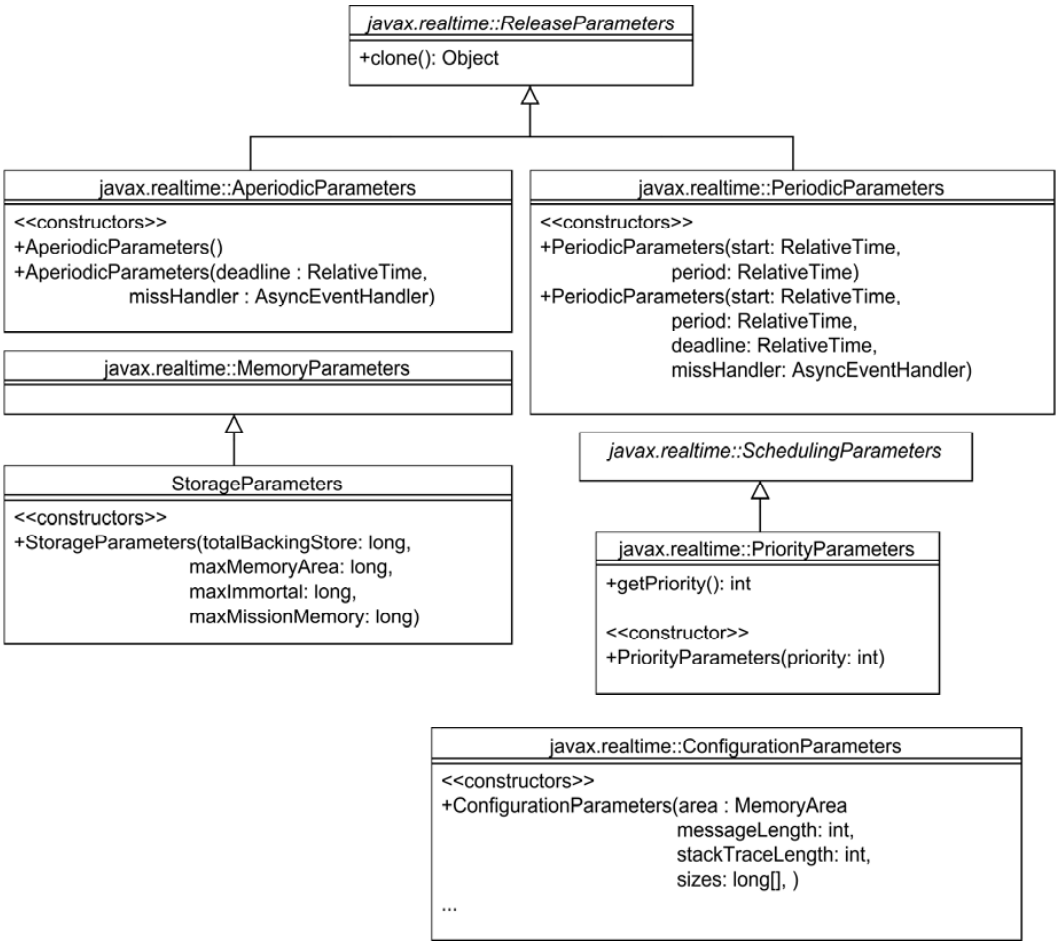

Figure 4.1: Parameter classes

- Calls to the Object.wait, Object.notify and Object.notifyAll methods are allowed. However, calling Object.wait from nested synchronized methods is illegal and shall result in raising an exeception if it is detected by the implementation.

4.4 The Parameter Classes

The run-time behaviors of SCJ schedulable objects are controlled by their associated parameter classes (see Figure 4.1):

- The ReleaseParameters class hierarchy — these enable the release characteristics of a schedulable object to be specified, for example whether it is periodic or aperiodic.

- The SchedulingParameters class hierarchy — these enable the priorities of the schedulable objects to be set.
- The MemoryParameters class hierarchy — these enable the amount of memory a schedulable object uses to be defined, including the amount of backing store needed for a schedulable object's private memories to be specified.
- The Configuration class — these enable the amount of memory that shall be allocated to the Java stack and for preallocated exceptions to be specified.
- ScopeParameters – this permits, among other things, the storage used by a schedulable object's scoped memory areas to be specified.

4.4.1 Class javax.realtime.ReleaseParameters

Declaration

```
@SCJAllowed
public abstract class ReleaseParameters implements java.lang.Cloneable,
    java.io.Serializable extends java.lang.Object
```

Description

This is the base class for the release parameters hierarchy. All schedulability analysis of safety critical software is performed by the application developers offline. Although the RTSJ allows on-line schedulability analysis, SCJ assumes any such analysis is performed off line and that the on-line environment is predictable. Consequently, the assumption is that deadlines are not missed. However, to facilitate fault-tolerant applications, SCJ does support a deadline miss detection facility at Level 1 and Level 2. SCJ provides no direct mechanisms for coping with cost overruns.

The ReleaseParameters class hierarchy is restricted so that the parameters can be set, but not changed or queried.

4.4.2 Class javax.realtime.PeriodicParameters

Declaration

```
@SCJAllowed
public class PeriodicParameters extends javax.realtime.ReleaseParameters
```

Description

This RTSJ class is restricted so that it allows the start time and the period to be set but not to be subsequently changed or queried.

Constructors

```
@SCJAllowed
@SCJMaySelfSuspend(false)
@SCJMayAllocate({})
@SCJPhase({
    javax.safetycritical.annotate.Phase.STARTUP,
    javax.safetycritical.annotate.Phase.INITIALIZATION,
    javax.safetycritical.annotate.Phase.RUN,
    javax.safetycritical.annotate.Phase.CLEANUP })
public PeriodicParameters(RelativeTime start,
  RelativeTime period,
  RelativeTime deadline,
  AsyncEventHandler missHandler)
```

Construct a new PeriodicParameters object within the current memory area.

start — is time of the first release of the associated schedulable object relative to the start of the mission. A null value defaults to an offset of zero milliseconds.

period — is the time between each release of the associated schedulable object.

deadline — is an offset from the release time by which the release should finish. A null deadline indicates the same value as the period.

missHandler — is the AperiodicEventHandler to be released if the associated schedulable object misses its deadline. A null parameter indicates that no handler should be released.

Throws IllegalArgumentException if the period is null or its time value is not greater than zero, or if the time value of deadline is not greater than zero, or if the clock associated with the start, period and deadline parameters is not the same.

```
@SCJAllowed
@SCJMaySelfSuspend(false)
@SCJMayAllocate({})
@SCJPhase({
    javax.safetycritical.annotate.Phase.STARTUP,
    javax.safetycritical.annotate.Phase.INITIALIZATION,
    javax.safetycritical.annotate.Phase.RUN,
    javax.safetycritical.annotate.Phase.CLEANUP })
public PeriodicParameters(RelativeTime start, RelativeTime period)
```

This constructor behaves the same as calling PeriodicParameters(start, period, null, null).

Methods

```
@SCJAllowed
@Override
```

```
@SCJMaySelfSuspend(false)
@SCJMayAllocate({javax.safetycritical.annotate.AllocationContext.CURRENT})
@SCJPhase({
   javax.safetycritical.annotate.Phase.STARTUP,
   javax.safetycritical.annotate.Phase.INITIALIZATION,
   javax.safetycritical.annotate.Phase.RUN,
   javax.safetycritical.annotate.Phase.CLEANUP })
public java.lang.Object clone( )
```

Create a clone of this PeriodicParameters object.

4.4.3 Class javax.realtime.AperiodicParameters

Declaration

```
@SCJAllowed(javax.safetycritical.annotate.Level.LEVEL_1)
 public class AperiodicParameters extends javax.realtime.ReleaseParameters
```

Description

SCJ supports no detection of minimum inter-arrival time violations, therefore only aperiodic parameters are needed. Hence the RTSJ SporadicParameters class is absent. Deadline miss detection is supported.

The RTSJ supports a queue for storing the arrival of release events is order to enable bursts of events to be handled. This queue is of length 1 in SCJ. The RTSJ also enables different responses to the queue overflowing. In SCJ the overflow behavior is to overwrite the pending release event if there is one.

Constructors

```
@SCJAllowed(javax.safetycritical.annotate.Level.LEVEL_1)
@SCJMaySelfSuspend(false)
@SCJMayAllocate({})
@SCJPhase({
   javax.safetycritical.annotate.Phase.STARTUP,
   javax.safetycritical.annotate.Phase.INITIALIZATION,
   javax.safetycritical.annotate.Phase.RUN,
   javax.safetycritical.annotate.Phase.CLEANUP })
public AperiodicParameters(RelativeTime deadline, AsyncEventHandler missHandler)
```

Construct a new AperiodicParameters object within the current memory area.

deadline — is an offset from the release time by which the release should finish. A null deadline indicates that there is no deadline.

missHandler — is the AsynchronousEventHandler to be released if the associated schedulable object misses its deadline. A null parameter indicates that no handler should be released.

```
@SCJAllowed(javax.safetycritical.annotate.Level.LEVEL_1)
@SCJMaySelfSuspend(false)
@SCJMayAllocate({})
@SCJPhase({
    javax.safetycritical.annotate.Phase.STARTUP,
    javax.safetycritical.annotate.Phase.INITIALIZATION,
    javax.safetycritical.annotate.Phase.RUN,
    javax.safetycritical.annotate.Phase.CLEANUP })
public AperiodicParameters( )
```

This constructor behaves the same as calling AperiodicParameters(null, null).

Methods

```
@SCJAllowed
@Override
@SCJMaySelfSuspend(false)
@SCJMayAllocate({javax.safetycritical.annotate.AllocationContext.CURRENT})
@SCJPhase({
    javax.safetycritical.annotate.Phase.STARTUP,
    javax.safetycritical.annotate.Phase.INITIALIZATION,
    javax.safetycritical.annotate.Phase.RUN,
    javax.safetycritical.annotate.Phase.CLEANUP })
public java.lang.Object clone( )
```

Create a clone of this AperiodicParameters object.

4.4.4 Class javax.realtime.SchedulingParameters

Declaration

```
@SCJAllowed
 public abstract class SchedulingParameters implements java.lang.Cloneable,
     java.io.Serializable extends java.lang.Object
```

Description

The RTSJ potentially allows different schedulers to be supported and defines this class as the root class for all scheduling parameters. In SCJ this class is empty; only priority parameters are supported.

There is no ImportanceParameters subclass in SCJ.

Methods

```
@SCJAllowed
@Override
@SCJMaySelfSuspend(false)
@SCJMayAllocate({javax.safetycritical.annotate.AllocationContext.CURRENT})
@SCJPhase({
    javax.safetycritical.annotate.Phase.STARTUP,
    javax.safetycritical.annotate.Phase.INITIALIZATION,
    javax.safetycritical.annotate.Phase.RUN,
    javax.safetycritical.annotate.Phase.CLEANUP })
public java.lang.Object clone( )
```

Create a clone of this SchedulingParameters object.

4.4.5 Class javax.realtime.PriorityParameters

Declaration

```
@SCJAllowed
 public class PriorityParameters extends javax.realtime.SchedulingParameters
```

Description

This class is restricted relative to the RTSJ so that it allows the priority to be created and queried, but not changed.

In SCJ the range of priorities is separated into software priorities and hardware priorities (see Section 4.7.5). Hardware priorities have higher values than software priorities. Schedulable objects can be assigned only software priorities. Ceiling priorities can be either software or hardware priorities.

Constructors

```
@SCJAllowed
@SCJMaySelfSuspend(false)
@SCJMayAllocate({})
@SCJPhase({
    javax.safetycritical.annotate.Phase.STARTUP,
    javax.safetycritical.annotate.Phase.INITIALIZATION,
    javax.safetycritical.annotate.Phase.RUN,
    javax.safetycritical.annotate.Phase.CLEANUP })
public PriorityParameters(int priority)
```

Create a PriorityParameters object specifying the given priority.

priority — is the integer value of the specified priority.

Throws IllegalArgumentException if priority is not in the range of supported priorities.

Methods

```
@SCJAllowed
@SCJMaySelfSuspend(false)
@SCJMayAllocate({})
@SCJPhase({
    javax.safetycritical.annotate.Phase.STARTUP,
    javax.safetycritical.annotate.Phase.INITIALIZATION,
    javax.safetycritical.annotate.Phase.RUN,
    javax.safetycritical.annotate.Phase.CLEANUP })
public int getPriority( )
```

returns the integer priority value that was specified at construction time.

4.4.6 Class javax.realtime.MemoryParameters

Declaration

```
@SCJAllowed
 public class MemoryParameters implements java.lang.Cloneable,
     java.io.Serializable extends java.lang.Object
```

Description

This class is used to define the maximum amount of memory that a schedulable object requires in its default memory area (its per-release private scope memory) and in immortal memory. The SCJ restricts this class relative to the RTSJ such that values can be created but not queried or changed.

Fields

```
@SCJAllowed
public static final long UNLIMITED
```

```
@SCJAllowed
public static final long UNREFERENCED
```

Constructors

```
@SCJAllowed
@SCJMaySelfSuspend(false)
@SCJMayAllocate({})
@SCJPhase({
    javax.safetycritical.annotate.Phase.STARTUP,
    javax.safetycritical.annotate.Phase.INITIALIZATION,
    javax.safetycritical.annotate.Phase.RUN,
    javax.safetycritical.annotate.Phase.CLEANUP })
public MemoryParameters(long maxInitialArea, long maxImmortal)
```

Create a MemoryParameters object with the given maximum values.

maxInitialArea — is the maximum amount of memory in the per-release private memory area.

maxImmortal — is the maximum amount of memory in the immortal memory area required by the associated schedulable object.

Throws IllegalArgumentException if any value is negative, or if NO_MAX is passed as the value of maxMemoryArea or maxImmortal.

Methods

```
@SCJAllowed
@Override
@SCJMaySelfSuspend(false)
@SCJMayAllocate({javax.safetycritical.annotate.AllocationContext.CURRENT})
@SCJPhase({
    javax.safetycritical.annotate.Phase.STARTUP,
    javax.safetycritical.annotate.Phase.INITIALIZATION,
    javax.safetycritical.annotate.Phase.RUN,
    javax.safetycritical.annotate.Phase.CLEANUP })
public java.lang.Object clone( )
```

Create a clone of this MemoryParameters object.

4.4.7 Class javax.realtime.ConfigurationParameters

Declaration

```
@SCJAllowed
 public class ConfigurationParameters extends java.lang.Object
```

Description

Schedulable sizing parameters a way to specify various implementation-dependent parameters such as Java and native stack sizes, and to configure the statically allocated ThrowBoundaryError associated with a Schedulable.

Note that these parameters are immutable.

Constructors

```
@SCJAllowed
@SCJPhase({
    javax.safetycritical.annotate.Phase.INITIALIZATION,
    javax.safetycritical.annotate.Phase.RUN,
    javax.safetycritical.annotate.Phase.CLEANUP})
@SCJMaySelfSuspend(false)
@SCJMayAllocate({})
public ConfigurationParameters(int messageLength,
   int stackTraceLength,
   long [] sizes)
```

Creates a parameter object for initializing the state of a Schedulable. The parameters provide the data for this initialization.

messageLength — Memory space in bytes dedicated to the message associated with Schedulable objects created with these parameters' preallocated exceptions, plus references to the method names/identifiers in the stack trace. The value 0 indicates that no message should be stored. The value of -1 uses the system default.

stackTraceLength — Length of the stack trace buffer dedicated to Schedulable objects created with these parameters' preallocated exceptions, in frames. The amount of space this requires is implementation-specific. The value 0 indicates that no stack trace should be stored. The value of -1 uses the system default.

sizes — An array of implementation-specific values dictating memory parameters for Schedulable objects created with these parameters, such as maximum Java and native stack sizes. The sizes array will not be stored in the constructed object.

Throws IllegalArgumentException if messageLength or stackTraceLength is less than -1.

4.4.8 Class javax.realtime.memory.ScopeParameters

Declaration

```
@SCJAllowed
 public class ScopeParameters extends javax.realtime.MemoryParameters
```

See Also: javax.realtime.MemoryParameters

Description

Extend memory parameters to provide limits for scoped memory.

Constructors

```
@SCJAllowed
@SCJMayAllocate({
   javax.safetycritical.annotate.AllocationContext.CURRENT,
   javax.safetycritical.annotate.AllocationContext.INNER,
   javax.safetycritical.annotate.AllocationContext.OUTER})
@SCJMaySelfSuspend(true)
@SCJPhase({
   javax.safetycritical.annotate.Phase.STARTUP,
   javax.safetycritical.annotate.Phase.INITIALIZATION,
   javax.safetycritical.annotate.Phase.RUN,
   javax.safetycritical.annotate.Phase.CLEANUP })
public ScopeParameters(long maxInitialArea,
  long maxImmortal,
  long maxContainingArea,
  long maxInitialBackingStore)
  throws java.lang.IllegalArgumentException
```

Create a ScopeParameters instance with the given values.

maxInitialArea — a limit on the amount of memory the schedulable may allocate in its initial scoped memory area. Units are in bytes. When zero, no allocation is allowed in the memory area. When the initial memory area is not a ScopedMemory , this parameter has no effect. To specify no limit, use MemoryParameters.UNLIMITED .

maxImmortal — A limit on the amount of memory the schedulable may allocate in the immortal area. Units are in bytes. When zero, no allocation allowed in immortal. To specify no limit, use MemoryParameters.UNLIMITED .

maxContainingArea — a limit on the amount of memory the schedulable may allocate in memory area where it was created. Units are in bytes. When zero, no allocation is allowed in the memory area. When the containing memory area is not a ScopedMemory , this parameter has no effect. To specify no limit, use MemoryParameters.UNLIMITED . For schedulables created within a mission, the containing memory area is Mission memory. For the initial MissionSequencer, the initial memory area is Immortal memory.

maxInitialBackingStore — A limit on the amount of backing store this task may allocate from backing store of its inital area, when that is a stacked memory. Units are in bytes. When zero, no allocation is allowed in that memory area. Backing store that is returned to the global backing store is subtracted from the limit. To specify no limit, use MemoryParameters.UNLIMITED .

Throws IllegalArgumentException when any value other than positive, zero, or **javax.realtime.MemoryParametersUNREFERENCED** is passed as the value of maxInitialArea , maxImmortal , maxParentBackingStore , or maxContainingArea.

```
@SCJAllowed
@SCJMayAllocate({
   javax.safetycritical.annotate.AllocationContext.CURRENT,
   javax.safetycritical.annotate.AllocationContext.INNER,
   javax.safetycritical.annotate.AllocationContext.OUTER})
@SCJMaySelfSuspend(true)
@SCJPhase({
   javax.safetycritical.annotate.Phase.STARTUP,
   javax.safetycritical.annotate.Phase.INITIALIZATION,
   javax.safetycritical.annotate.Phase.RUN,
   javax.safetycritical.annotate.Phase.CLEANUP })
public ScopeParameters(long maxInitialArea,
  long maxImmortal,
  long maxInitialBackingStore)
  throws java.lang.IllegalArgumentException
```

Same as ScopeParameters(maxInitialArea, maxImmortal, maxParentBackingStore, MemoryParameters.UNLIMITED).

Methods

```
@SCJAllowed
@SCJMaySelfSuspend(false)
@SCJPhase({
    javax.safetycritical.annotate.Phase.INITIALIZATION,
    javax.safetycritical.annotate.Phase.RUN,
    javax.safetycritical.annotate.Phase.CLEANUP})
@SCJMayAllocate({})
public long getMaxBackingStore( )
```

Determine the limit on backing store for this task.

returns the limit on backing store.

```
@SCJMaySelfSuspend(false)
@SCJPhase({
    javax.safetycritical.annotate.Phase.INITIALIZATION,
    javax.safetycritical.annotate.Phase.RUN,
    javax.safetycritical.annotate.Phase.CLEANUP})
@SCJMayAllocate({})
@SCJAllowed
public long getMaxContainingArea( )
```

Determine the limit on allocation in the area where the task was created.

returns the limit on allocation in the area where the task was created.

4.5 Asynchronous Event Handlers

The event based programming paradigm in SCJ (see Figure 4.2) may be implemented using the RTSJ asynchronous event handling mechanisms. The types of event handlers are very constrained in SCJ relative to the corresponding classes in the RTSJ. Consequently, SCJ defines a set of new subclasses to support them. Therefore, direct use of the RTSJ classes by the application is disallowed.

In SCJ all explicit application use of asynchronous events is hidden by the SCJ infrastructure. The SCJ API provides only handler definitions. Where the handlers are time-triggered, the SCJ classes allow the timing requirements to be passed through the constructors and, where appropriate, to be queried and reset etc. Where the handlers are event triggered, the SCJ classes provide a release mechanism.

The class hierarchy that supports the SCJ model is given in the remainder of this section and illustrated in Figure 4.2. Discussion of the integration of this model with POSIX signal handling is deferred until the next Chapter.

Unless indicated otherwise, the classes defined in this section are thread safe

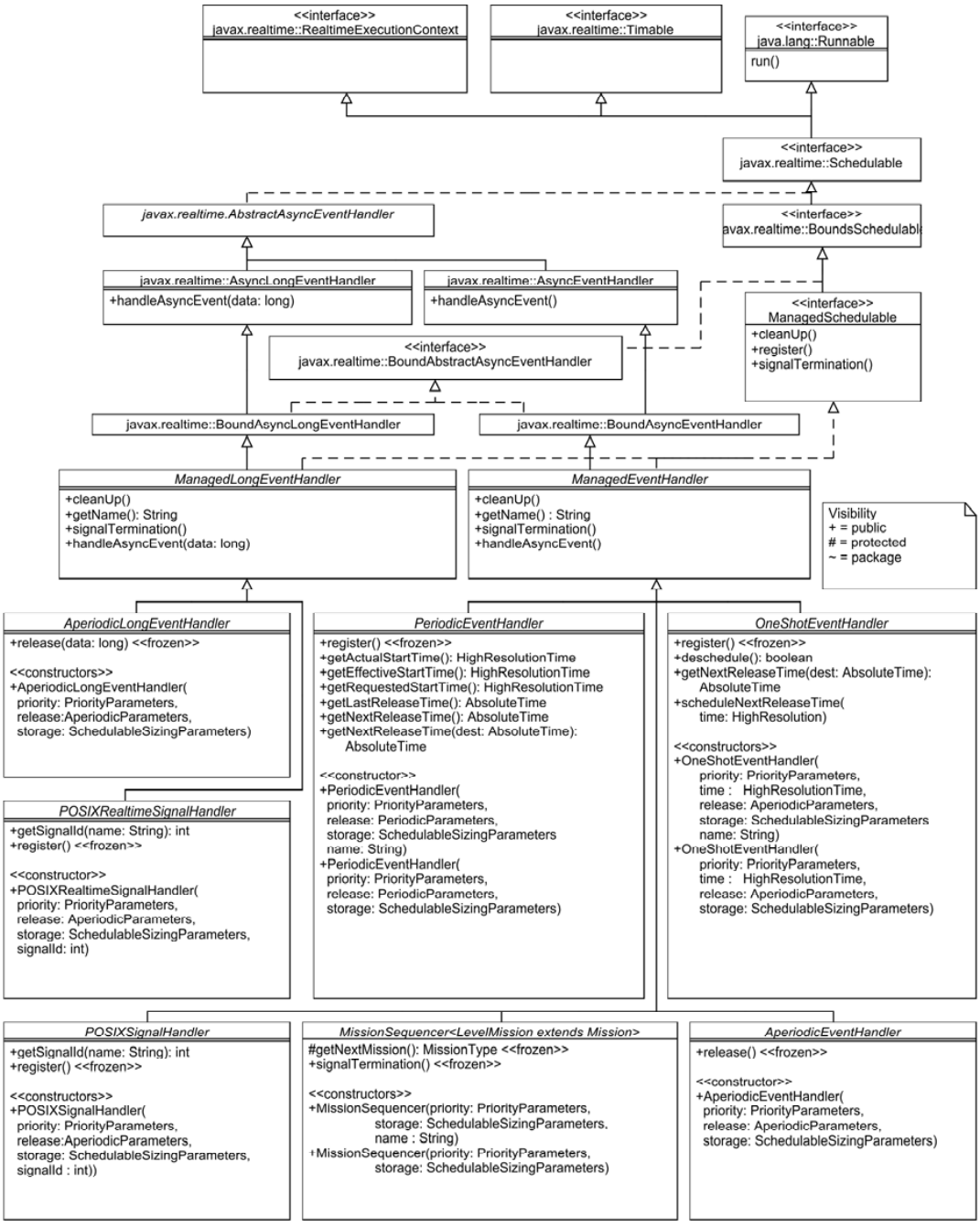

Figure 4.2: Abridged Handler classes

4.5.1 Interface javax.realtime.Timable

Declaration

```
@SCJAllowed
 public interface Timable extends javax.realtime.Releasable
```

Description

An interface for RealtimeThread to indicate that it can be associated with a clock and be suspended waiting for timing events based on that clock. This interface make use of some interfaces and classes in the RTSJ that are not visible to the SCJ. They are, therefore, not presented in this specification.

4.5.2 Interface javax.realtime.AsyncTimable

Declaration

```
@SCJAllowed
 public interface AsyncTimable extends javax.realtime.Timable
```

Description

An interface to indicate it they can be associated with a Clock and be suspended waiting for time events based on that clock.

4.5.3 Interface javax.realtime.Schedulable

Declaration

```
@SCJAllowed
 public interface Schedulable extends java.lang.Runnable, javax.realtime.Timable
```

Description

In keeping with the RTSJ, SCJ event handlers are schedulable objects. However, the Schedulable interface in the RTSJ is mainly concerned with the getting and setting of the parameter classes. On the contrary, in SCJ, these facilities are not provided. All that is supported in the methods that allow schedulable objects to be interrupted.

Methods

```
@SCJAllowed
@SCJPhase({javax.safetycritical.annotate.Phase.RUN})
@SCJMayAllocate({})
@SCJMaySelfSuspend(false)
public void interrupt( )
```

Behaves as if {@code Thread.interrupt()} were called on the implementation thread underlying this Schedulable. throws IllegalSchedulableStateException when {@code this} is not currently releasable, i.e., its start method has not been called, or it has terminated.

```
@SCJAllowed
@SCJPhase({javax.safetycritical.annotate.Phase.RUN})
@SCJMayAllocate({})
@SCJMaySelfSuspend(false)
public boolean isInterrupted( )
```

Determines whether or not any interrupt is pending.

returns {@code true} when and only when the interrupt is pending.

4.5.4 Interface javax.realtime.BoundRealtimeExecutor

Declaration

```
@SCJAllowed
 public interface BoundRealtimeExecutor
```

Description

This interface denotes all RTSJ and SCJ objects that encapsulate execution. In SCJ, this type includes Schedulable and InterruptServiceRoutine objects. It is used by Affinity to remove the need to have a reference into the javax.realtime.device package.

Methods

```
@SCJMaySelfSuspend(false)
@SCJPhase({
   javax.safetycritical.annotate.Phase.STARTUP,
   javax.safetycritical.annotate.Phase.INITIALIZATION,
   javax.safetycritical.annotate.Phase.RUN,
   javax.safetycritical.annotate.Phase.CLEANUP})
@SCJMayAllocate({})
@SCJAllowed(javax.safetycritical.annotate.Level.LEVEL_1)
public javax.realtime.Affinity getAffinity( )
```

Determine the affinity set instance associated with {@code task}.

returns The associated affinity.

```
@SCJMaySelfSuspend(false)
@SCJPhase({
   javax.safetycritical.annotate.Phase.STARTUP,
   javax.safetycritical.annotate.Phase.INITIALIZATION})
@SCJMayAllocate({})
@SCJAllowed(javax.safetycritical.annotate.Level.LEVEL_1)
public void setAffinity(Affinity set)
  throws java.lang.IllegalArgumentException,
     javax.realtime.ProcessorAffinityException, java.lang.NullPointerException
```

Set the processor affinity of a {@code task} to {@code set} with immediate effect.

set — is the processor affinity

Throws IllegalArgumentException when the intersection of {@code set} the affinity of any {@code ThreadGroup} instance containing {@code task} is empty.

Throws ProcessorAffinityException is thrown when the runtime fails to set the affinity for platform-specific reasons.

Throws NullPointerException when {@code set} is {@code null}.

4.5.5 Interface javax.realtime.BoundSchedulable

Declaration

```
@SCJAllowed
 public interface BoundSchedulable extends javax.realtime.Schedulable,
     javax.realtime.BoundRealtimeExecutor
```

Description

A marker interface to provide a type safe reference to all schedulables that are bound to a single underlying thread.

4.5.6 Interface javax.safetycritical.ManagedSchedulable

Declaration

```
@SCJAllowed
 public interface ManagedSchedulable extends javax.realtime.BoundSchedulable
```

Description

In SCJ, all schedulable objects are managed by a mission.

This interface is implemented by all SCJ Schedulable classes. It defines the mechanism by which the ManagedSchedulable is registered with the mission for its management. This interface is used by SCJ classes. It is not intended for direct use by application classes.

Methods

```
@SCJAllowed(javax.safetycritical.annotate.Level.SUPPORT)
@SCJPhase({javax.safetycritical.annotate.Phase.CLEANUP})
@SCJMaySelfSuspend(false)
@SCJMayAllocate({
   javax.safetycritical.annotate.AllocationContext.CURRENT,
   javax.safetycritical.annotate.AllocationContext.INNER,
   javax.safetycritical.annotate.AllocationContext.OUTER})
public void cleanUp( )
```

Runs any end-of-mission clean up code associated with this schedulable object.

Application developers implement this method with code to be executed when this schedulable object's execution is disabled (after termination has been requested of the enclosing mission).

When the cleanUp method is called, the private memory area associated with this schedulable object shall be the current memory area. If desired, the cleanUp method may introduce new private memory areas. The memory allocated to ManagedSchedulables shall be available to be reclaimed when its Mission's cleanUp method returns.

```
@SCJAllowed
@SCJPhase({javax.safetycritical.annotate.Phase.INITIALIZATION})
@SCJMayAllocate({})
@SCJMaySelfSuspend(false)
public void register( )
```

Register this managed schedulable with the current mission.

```
@SCJAllowed(javax.safetycritical.annotate.Level.SUPPORT)
@SCJMayAllocate({})
@SCJMaySelfSuspend(false)
@SCJPhase({
   javax.safetycritical.annotate.Phase.RUN,
   javax.safetycritical.annotate.Phase.CLEANUP})
public void signalTermination( )
```

Called by the infrastructure to indicate that the enclosing mission has been instructed to terminate.

The application can override the default implementations of signalTermination() to facilitate termination of the ManagedSchedulable.

4.5.7 Class javax.realtime.AsyncBaseEventHandler

Declaration

```
@SCJAllowed
 public abstract class AsyncBaseEventHandler implements
    javax.realtime.Schedulable extends java.lang.Object
```

Description

This is the base class for all asynchronous event handlers. In SCJ, this is an empty class.

Methods

```
@SCJAllowed
@SCJPhase({javax.safetycritical.annotate.Phase.RUN})
@SCJMayAllocate({})
@SCJMaySelfSuspend(false)
public void interrupt( )
```

Behaves as if {@code Thread.interrupt()} were called on the implementation thread underlying this Schedulable. throws IllegalSchedulableStateException when {@code this} is not currently releasable, i.e., its start method has not been called, or it has terminated.

```
@SCJAllowed
@SCJPhase({javax.safetycritical.annotate.Phase.RUN})
@SCJMayAllocate({})
@SCJMaySelfSuspend(false)
public boolean isInterrupted( )
```

Determines whether or not any interrupt is pending.

returns {@code true} when and only when the interrupt is pending.

4.5.8 Class javax.realtime.AsyncEventHandler

Declaration

```
@SCJAllowed
 public class AsyncEventHandler extends javax.realtime.AsyncBaseEventHandler
```

Description

In SCJ, all asynchronous events have their handlers bound to a thread when they are created (during the initialization phase). The binding is permanent. Thus, the AsyncEventHandler constructors are hidden from public view in the SCJ specification.

Methods

```
@SCJAllowed(javax.safetycritical.annotate.Level.SUPPORT)
@SCJPhase({javax.safetycritical.annotate.Phase.RUN})
@SCJMayAllocate({
   javax.safetycritical.annotate.AllocationContext.CURRENT,
   javax.safetycritical.annotate.AllocationContext.INNER,
   javax.safetycritical.annotate.AllocationContext.OUTER})
@SCJMaySelfSuspend(true)
public void handleAsyncEvent( )
```

This method must be overridden by the application to provide the handling code. Note that this method shall not self-suspend when called in a Level 0 mission.

```
@SCJAllowed(javax.safetycritical.annotate.Level.SUPPORT)
@SCJPhase({javax.safetycritical.annotate.Phase.RUN})
@Override
@SCJMayAllocate({
   javax.safetycritical.annotate.AllocationContext.CURRENT,
   javax.safetycritical.annotate.AllocationContext.INNER,
   javax.safetycritical.annotate.AllocationContext.OUTER})
@SCJMaySelfSuspend(true)
public final void run( )
```

This method is used by the SCJ infrastructure. It should not be called by the application.

4.5.9 Class javax.realtime.AsyncLongEventHandler

Declaration

```
@SCJAllowed
 public class AsyncLongEventHandler extends javax.realtime.AsyncBaseEventHandler
```

Description

In SCJ, all asynchronous events must have their handlers bound when they are created (during the initialization phase). The binding is permanent. Thus, the AsyncLongEventHandler constructors are hidden from public view in the SCJ specification. This class differs from AsyncEventHandler in that when it is fired, a long integer is provided for use by the released event handler(s).

Methods

```
@SCJAllowed(javax.safetycritical.annotate.Level.SUPPORT)
@SCJPhase({javax.safetycritical.annotate.Phase.RUN})
@SCJMayAllocate({
   javax.safetycritical.annotate.AllocationContext.CURRENT,
   javax.safetycritical.annotate.AllocationContext.INNER,
   javax.safetycritical.annotate.AllocationContext.OUTER})
@SCJMaySelfSuspend(true)
public void handleAsyncEvent(long data)
```

This method must be overridden by the application to provide the handling code. Note that this method shall not self-suspend when called in a Level 0 mission.

```
@SCJAllowed(javax.safetycritical.annotate.Level.SUPPORT)
@SCJPhase({javax.safetycritical.annotate.Phase.RUN})
@Override
@SCJMayAllocate({
   javax.safetycritical.annotate.AllocationContext.CURRENT,
   javax.safetycritical.annotate.AllocationContext.INNER,
   javax.safetycritical.annotate.AllocationContext.OUTER})
@SCJMaySelfSuspend(true)
public final void run( )
```

This method is used by the SCJ infrastructure. It should not be called by the application.

4.5.10 Interface javax.realtime.BoundAsyncBaseEventHandler

Declaration

```
@SCJAllowed
 public interface BoundAsyncBaseEventHandler extends
     javax.realtime.BoundSchedulable
```

Description

An empty interface. It is required in order to allow references to all bound handlers.

4.5.11 Class javax.realtime.BoundAsyncEventHandler

Declaration

```
@SCJAllowed
 public class BoundAsyncEventHandler implements
     javax.realtime.BoundAsyncBaseEventHandler extends
     javax.realtime.AsyncEventHandler
```

Description

The BoundAsyncEventHandler class is a base class inherited from RTSJ. None of its methods or constructors are publicly available.

Methods

```
@SCJAllowed(javax.safetycritical.annotate.Level.LEVEL_1)
@SCJMaySelfSuspend(false)
@SCJPhase({
   javax.safetycritical.annotate.Phase.STARTUP,
   javax.safetycritical.annotate.Phase.INITIALIZATION,
   javax.safetycritical.annotate.Phase.RUN,
   javax.safetycritical.annotate.Phase.CLEANUP})
@SCJMayAllocate({})
@Override
public javax.realtime.Affinity getAffinity( )


@SCJAllowed(javax.safetycritical.annotate.Level.LEVEL_1)
@SCJMaySelfSuspend(false)
@SCJPhase({
   javax.safetycritical.annotate.Phase.STARTUP,
   javax.safetycritical.annotate.Phase.INITIALIZATION})
@SCJMayAllocate({})
@Override
public void setAffinity(Affinity set)
  throws java.lang.IllegalArgumentException,
     javax.realtime.ProcessorAffinityException, java.lang.NullPointerException
```

4.5.12 Class javax.realtime.BoundAsyncLongEventHandler

Declaration

```
@SCJAllowed
 public class BoundAsyncLongEventHandler implements
     javax.realtime.BoundAsyncBaseEventHandler extends
     javax.realtime.AsyncLongEventHandler
```

Description

The BoundAsyncLongEventHandler is a base class inherited from RTSJ. None of its methods or constructors are publicly available. This class differs from BoundAsyncEventHandler in that when it is released, a long integer is provided for use by the released event handler(s).

Methods

```
@SCJAllowed(javax.safetycritical.annotate.Level.LEVEL_1)
@SCJMaySelfSuspend(false)
```

```
@SCJPhase({
   javax.safetycritical.annotate.Phase.STARTUP,
   javax.safetycritical.annotate.Phase.INITIALIZATION})
@SCJMayAllocate({})
@Override
public javax.realtime.Affinity getAffinity( )
```

Note: since this is only used by infrastructure, we don't specify the MemoryAreaEncloses relationships. public BoundAsyncLongEventHandler(SchedulingParameters scheduling, ReleaseParameters release, MemoryParameters memory, MemoryArea area, ProcessingGroupParameters group, boolean noheap, Runnable logic) { super(scheduling, release, memory, area, group, noheap, logic); }

```
@SCJAllowed(javax.safetycritical.annotate.Level.LEVEL_1)
@SCJMaySelfSuspend(false)
@SCJPhase({
   javax.safetycritical.annotate.Phase.STARTUP,
   javax.safetycritical.annotate.Phase.INITIALIZATION,
   javax.safetycritical.annotate.Phase.RUN,
   javax.safetycritical.annotate.Phase.CLEANUP})
@SCJMayAllocate({})
@Override
public void setAffinity(Affinity set)
  throws java.lang.IllegalArgumentException,
     javax.realtime.ProcessorAffinityException, java.lang.NullPointerException
```

4.5.13 Class javax.safetycritical.ManagedEventHandler

Declaration

```
@SCJAllowed
 public abstract class ManagedEventHandler implements
     javax.safetycritical.ManagedSchedulable extends
     javax.realtime.BoundAsyncEventHandler
```

Description

In SCJ, all handlers must be registered with the enclosing mission, so SCJ applications use classes that are based on the ManagedEventHandler and the ManagedLongEventHandler class hierarchies. These class hierarchies allow a mission to manage all the handlers that are created during its initialization phase. The infrastructure sets up a private memory area for each managed handler that is entered before a call to handleAsyncEvent and is left on return.

The scheduling allocation domain of all managed event handlers is set, by default, to the scheduling allocation domain from where the associated mission initialization is being performed.

Methods

```
@SCJAllowed(javax.safetycritical.annotate.Level.SUPPORT)
@SCJPhase({javax.safetycritical.annotate.Phase.CLEANUP})
@SCJMaySelfSuspend(false)
@Override
@SCJMayAllocate({
   javax.safetycritical.annotate.AllocationContext.CURRENT,
   javax.safetycritical.annotate.AllocationContext.INNER,
   javax.safetycritical.annotate.AllocationContext.OUTER})
public void cleanUp( )
```

Runs any end-of-mission clean up code associated with this schedulable object.
see ManagedSchedulable.cleanUp()

```
@SCJAllowed
@SCJMaySelfSuspend(false)
@SCJMayAllocate({})
@SCJPhase({
   javax.safetycritical.annotate.Phase.STARTUP,
   javax.safetycritical.annotate.Phase.INITIALIZATION,
   javax.safetycritical.annotate.Phase.RUN,
   javax.safetycritical.annotate.Phase.CLEANUP })
public java.lang.String getName( )
```

returns a string name of this event handler. The actual object returned shall be the same object that was passed to the event handler constructor.

```
@SCJPhase({javax.safetycritical.annotate.Phase.INITIALIZATION})
@SCJMaySelfSuspend(false)
@SCJMayAllocate({})
@Override
@SCJAllowed
public void register( )
```

Register this event handler with the current mission.

see javax.safetycritical.ManagedSchedulable.register()

```
@SCJAllowed(javax.safetycritical.annotate.Level.SUPPORT)
@SCJMaySelfSuspend(false)
@SCJMayAllocate({})
@SCJPhase({
   javax.safetycritical.annotate.Phase.RUN,
   javax.safetycritical.annotate.Phase.CLEANUP})
public void signalTermination( )
```

Called by the infrastructure to indicate that the enclosing mission has been instructed to terminate. The default behavior is no action.

4.5.14 Class javax.safetycritical.ManagedLongEventHandler

Declaration

```
@SCJAllowed
 public abstract class ManagedLongEventHandler implements
     javax.safetycritical.ManagedSchedulable extends
     javax.realtime.BoundAsyncLongEventHandler
```

Description

In SCJ, all handlers must be registered with the enclosing mission, so SCJ applications use classes that are based on the ManagedEventHandler and the ManagedLongEventHandler class hierarchies. These class hierarchies allow a mission to manage all the handlers that are created during its initialization phase. The infrastructure sets up a private memory area for each managed handler that is entered before a call to handleAsyncEvent and is left on return. This class differs from ManagedEventHandler in that when it is released, a long integer is provided for use by the released event handler(s).

The scheduling allocation domain of all managed long event handlers is set, by default, to the scheduling allocation domain from where the associated mission initialization is being performed.

Methods

```
@SCJAllowed(javax.safetycritical.annotate.Level.SUPPORT)
@SCJPhase({javax.safetycritical.annotate.Phase.CLEANUP})
@SCJMaySelfSuspend(false)
@Override
@SCJMayAllocate({
   javax.safetycritical.annotate.AllocationContext.CURRENT,
   javax.safetycritical.annotate.AllocationContext.INNER,
   javax.safetycritical.annotate.AllocationContext.OUTER})
public void cleanUp( )
```

Runs any end-of-mission clean up code associated with this schedulable object. see ManagedSchedulable.cleanUp()

```
@SCJAllowed
@SCJMaySelfSuspend(false)
@SCJMayAllocate({})
@SCJPhase({
   javax.safetycritical.annotate.Phase.STARTUP,
   javax.safetycritical.annotate.Phase.INITIALIZATION,
   javax.safetycritical.annotate.Phase.RUN,
   javax.safetycritical.annotate.Phase.CLEANUP })
public java.lang.String getName( )
```

returns a string name for this handler, including its priority and its level.

```
@SCJPhase({javax.safetycritical.annotate.Phase.INITIALIZATION})
@SCJMaySelfSuspend(false)
@SCJMayAllocate({})
@SCJAllowed
public void register( )
```

Register this event handler with the current mission.

See javax.safetycritical.ManagedSchedulable.register()

Throws IllegalStateException if this is an instance of MissionSequencer and the current execution environment does not support Level 2 capabilities.

```
@SCJAllowed(javax.safetycritical.annotate.Level.SUPPORT)
@SCJMaySelfSuspend(false)
@SCJMayAllocate({})
@SCJPhase({
   javax.safetycritical.annotate.Phase.RUN,
   javax.safetycritical.annotate.Phase.CLEANUP})
public void signalTermination( )
```

Called by the infrastructure to indicate that the enclosing mission has been instructed to terminate. The default behaviour is to perform no action.

4.5.15 Class javax.safetycritical.PeriodicEventHandler

Declaration

```
@SCJAllowed
 public abstract class PeriodicEventHandler extends
     javax.safetycritical.ManagedEventHandler
```

Description

This class permits the automatic periodic execution of code. The handleAsyncEvent method behaves as if the handler were attached to a periodic timer asynchronous event. The handler will be executed once for every release time, even in the presence of overruns.

This class is abstract; non-abstract sub-classes must override the method handleAsyncEvent and may override the default cleanUp method.

Note that the values in parameters passed to the constructors are those that will be used by the infrastructure. Changing these values after construction will have no impact on the created event handler.

Note: all time-triggered events are subject to release jitter. See section 4.8.4 for a discussion of the impact of this on application scheduling.

Constructors

```
@SCJAllowed(javax.safetycritical.annotate.Level.LEVEL_1)
@SCJPhase({javax.safetycritical.annotate.Phase.INITIALIZATION})
@SCJMaySelfSuspend(false)
@SCJMayAllocate({
    javax.safetycritical.annotate.AllocationContext.CURRENT,
    javax.safetycritical.annotate.AllocationContext.INNER,
    javax.safetycritical.annotate.AllocationContext.OUTER})
public PeriodicEventHandler(PriorityParameters priority,
  PeriodicParameters release,
  ScopeParameters storage,
  ConfigurationParameters config,
  String name)
```

Constructs a periodic event handler.

The values passed as constructor parameters are captured at construction time. Any subsequent mutation of the parameter objects has no effect on the behavior of the constructed object.

priority — specifies the priority parameters for this periodic event handler. Must not be null.

release — specifies the periodic release parameters, in particular the start time, period and deadline miss handler. Note that a relative start time is not relative to NOW but relative to the point in time when initialization is finished and the timers are started. If the start time is absolute and it is has passed, the handler is release immediately. This argument must not be null.

storage — specifies the ScopeParameters for this periodic event handler

config — specifies the ConfigurationParameters for this periodic event handler

Throws IllegalArgumentException when priority, release, or storage is null or when any deadline miss handler specified in release is not an AperiodicEventHandler.

```
@SCJAllowed
@SCJPhase({javax.safetycritical.annotate.Phase.INITIALIZATION})
@SCJMaySelfSuspend(false)
@SCJMayAllocate({
    javax.safetycritical.annotate.AllocationContext.CURRENT,
    javax.safetycritical.annotate.AllocationContext.INNER,
    javax.safetycritical.annotate.AllocationContext.OUTER})
public PeriodicEventHandler(PriorityParameters priority,
  PeriodicParameters release,
  ScopeParameters storage,
  ConfigurationParameters config)
```

This constructor behaves the same as a call to PeriodicEventHandler(PriorityParameters, PeriodicParameters, ConfigurationParameters, String) with the arguments (priority, release, storage, config, null).

Methods

```
@SCJAllowed(javax.safetycritical.annotate.Level.LEVEL_1)
@SCJPhase({
   javax.safetycritical.annotate.Phase.RUN,
   javax.safetycritical.annotate.Phase.CLEANUP})
@SCJMayAllocate({javax.safetycritical.annotate.AllocationContext.CURRENT})
@SCJMaySelfSuspend(false)
public <T extends javax.realtime.HighResolutionTime<T>> T getActualStartTime( )
```

Get the actual start time of this handler. The actual execution start time of the handler is different from the requested start time (passed at construction time) when the requested start time is an absolute time that would occur before the mission has been started. In this case, the actual start time is the time the mission started execution. If the actual start time is equal to the effective start time, then the method behaves as if getRequestedStartTime() method has been called. If it is different, then a newly created time object is returned. The time value is associated with the same clock as that used with the original start time parameter.

returns a reference to the actual start time based on the clock used to start the timer.

```
@SCJAllowed(javax.safetycritical.annotate.Level.LEVEL_1)
@SCJPhase({
   javax.safetycritical.annotate.Phase.RUN,
   javax.safetycritical.annotate.Phase.CLEANUP})
@SCJMayAllocate({javax.safetycritical.annotate.AllocationContext.CURRENT})
@SCJMaySelfSuspend(false)
public <T extends javax.realtime.HighResolutionTime<T>> T getEffectiveStartTime( )
```

Get the effective start time of this handler. If the clock associated with the start time parameter and the period parameter (that were passed at construction time) are the same, then the method behaves as if getActualStartTime() has been called. If the two clocks are different, then the method returns a newly created object whose time is the current time of the clock associated with the period parameter (passed at construction time) when the handler is actually started.

returns a reference to a newly-created object containing the effective start time based on the clock associated with the period parameter.

```
@SCJAllowed(javax.safetycritical.annotate.Level.LEVEL_1)
@SCJPhase({
    javax.safetycritical.annotate.Phase.RUN,
    javax.safetycritical.annotate.Phase.CLEANUP})
@SCJMayAllocate({javax.safetycritical.annotate.AllocationContext.CURRENT})
@SCJMaySelfSuspend(false)
public javax.realtime.AbsoluteTime getLastReleaseTime( )
```

Get the last release time of this handler.

returns a reference to a newly-created object containing this handlers's last release time, based on the clock associated with the period parameter used at construction time.

Throws IllegalStateException if this handler has not been released since it was started.

```
@SCJAllowed(javax.safetycritical.annotate.Level.LEVEL_1)
@SCJPhase({
    javax.safetycritical.annotate.Phase.RUN,
    javax.safetycritical.annotate.Phase.CLEANUP})
@SCJMayAllocate({javax.safetycritical.annotate.AllocationContext.CURRENT})
@SCJMaySelfSuspend(false)
public javax.realtime.AbsoluteTime getNextReleaseTime( )
```

Get the time at which this handler is next expected to be released.

returns The absolute time at which this handler is expected to be released in a newly allocated AbsoluteTime object. The clock association of the returned time is the clock on which the period parameter (passed at construction time) is based.

Throws IllegalStateException if this handler has not been started or has terminated.

```
@SCJAllowed(javax.safetycritical.annotate.Level.LEVEL_1)
@SCJPhase({
    javax.safetycritical.annotate.Phase.RUN,
    javax.safetycritical.annotate.Phase.CLEANUP})
@SCJMayAllocate({javax.safetycritical.annotate.AllocationContext.CURRENT})
@SCJMaySelfSuspend(false)
public javax.realtime.AbsoluteTime getNextReleaseTime(AbsoluteTime dest)
```

Get the time at which this handler is next expected to be released.

dest — The instance of AbsoluteTime which will be updated in place and returned. The clock association of the dest parameter is ignored. When dest is null, a new object is allocated for the result.

returns The instance of AbsoluteTime passed as parameter, containing the absolute time at which this handler is expected to be released. If the dest parameter is null the result is returned in a newly allocated object. The clock association of the returned time is the clock on which the period parameter (passed at construction time) is based.

Throws IllegalStateException if this handler has not been started or has terminated.

```
@SCJAllowed(javax.safetycritical.annotate.Level.LEVEL_1)
@SCJMaySelfSuspend(false)
@SCJMayAllocate({})
@SCJPhase({
    javax.safetycritical.annotate.Phase.STARTUP,
    javax.safetycritical.annotate.Phase.INITIALIZATION,
    javax.safetycritical.annotate.Phase.RUN,
    javax.safetycritical.annotate.Phase.CLEANUP })
public <T extends javax.realtime.HighResolutionTime<T>> T getRequestedStartTime( )
```

Get the requested start time of this periodic handler. Note that the start time uses copy semantics, so changes made to the value returned by this method will not affect the start time of this handler if it has not already been started.

returns a reference to the start time parameter from the release parameters used when constructing this handler.

4.5.16 Class javax.safetycritical.OneShotEventHandler

Declaration

```
@SCJAllowed(javax.safetycritical.annotate.Level.LEVEL_1)
 public abstract class OneShotEventHandler extends
     javax.safetycritical.ManagedEventHandler
```

Description

This class permits the automatic execution of time-triggered code. The handleAsyncEvent method behaves as if the handler were attached to a one-shot timer asynchronous event.

This class is abstract, non-abstract sub-classes must implement the method handleAsyncEvent and may override the default cleanUp method.

Note that the values in parameters passed to the constructors are those that will be used by the infrastructure. Changing these values after construction will have no impact on the created event handler.

Note: all time-triggered events are subject to release jitter. See section 4.8.4 for a discussion of the impact of this on application scheduling.

Constructors

```
@SCJAllowed
@SCJPhase({javax.safetycritical.annotate.Phase.INITIALIZATION})
@SCJMaySelfSuspend(false)
@SCJMayAllocate({
    javax.safetycritical.annotate.AllocationContext.CURRENT,
    javax.safetycritical.annotate.AllocationContext.INNER,
    javax.safetycritical.annotate.AllocationContext.OUTER})
public OneShotEventHandler(PriorityParameters priority,
  HighResolutionTime<?> time,
  AperiodicParameters release,
  ScopeParameters storage,
  ConfigurationParameters config,
  String name)
```

Constructs a one-shot event handler.

priority — specifies the priority parameters for this event handler. Must not be null.

time — specifies the time at which the handler should be released for execution. A relative time is relative to the start of the associated mission. An absolute time that is before the mission is started is equivalent to a relative time of 0.0. A null parameter indicates that no release of the handler should be scheduled.

release — specifies the aperiodic release parameters, in particular the deadline miss handler. A null parameter indicates that there is no deadline associated with this handler.

storage — specifies the ScopeParameters for this handler

config — specifies the ConfigurationParameters for this handler

name — a name provided by the application to be attached to this handler.

Throws IllegalArgumentException if priority or storage is null; or if time is a negative relative time; or when a deadline miss handler specfied in release is not an AperiodicEventHandler.

```
@SCJAllowed(javax.safetycritical.annotate.Level.LEVEL_1)
@SCJPhase({javax.safetycritical.annotate.Phase.INITIALIZATION})
@SCJMaySelfSuspend(false)
@SCJMayAllocate({
    javax.safetycritical.annotate.AllocationContext.CURRENT,
    javax.safetycritical.annotate.AllocationContext.INNER,
    javax.safetycritical.annotate.AllocationContext.OUTER})
public OneShotEventHandler(PriorityParameters priority,
  HighResolutionTime<?> time,
  AperiodicParameters release,
  ScopeParameters storage,
  ConfigurationParameters config)
```

This constructor behaves the same as a call to OneShotEventHandler(priority, time, release, storage, config, null).

```
@SCJAllowed(javax.safetycritical.annotate.Level.LEVEL_1)
@SCJPhase({javax.safetycritical.annotate.Phase.INITIALIZATION})
@SCJMaySelfSuspend(false)
@SCJMayAllocate({})
public OneShotEventHandler(PriorityParameters priority,
  AperiodicParameters release,
  ScopeParameters storage,
  ConfigurationParameters config)
```

This constructor behaves the same as a call to OneShotEventHandler(priority, null, release, storage, null).

Methods

```
@SCJAllowed(javax.safetycritical.annotate.Level.LEVEL_1)
@SCJPhase({
   javax.safetycritical.annotate.Phase.RUN,
   javax.safetycritical.annotate.Phase.CLEANUP})
@SCJMaySelfSuspend(false)
@SCJMayAllocate({})
public boolean deschedule( )
```

Deschedules the next release of the handler.

returns true if the handler was scheduled to be released false otherwise.

```
@SCJAllowed(javax.safetycritical.annotate.Level.LEVEL_1)
@SCJPhase({
   javax.safetycritical.annotate.Phase.RUN,
   javax.safetycritical.annotate.Phase.CLEANUP})
@SCJMaySelfSuspend(false)
@SCJMayAllocate({javax.safetycritical.annotate.AllocationContext.CURRENT})
public javax.realtime.AbsoluteTime getNextReleaseTime(AbsoluteTime dest)
```

Get the time at which this handler is next expected to be released.

dest — The instance of AbsoluteTime which will be updated in place and returned. The clock association of the dest parameter is ignored. When dest is null a new object is allocated for the result.

returns An instance of an AbsoluteTime object containing the absolute time at which this handler is expected to be released, or null if there is no currently scheduled release. If the dest parameter is null the result is returned in a newly allocated object; otherwise it is returned in the dest object.

```
@SCJAllowed(javax.safetycritical.annotate.Level.LEVEL_1)
@SCJPhase({
   javax.safetycritical.annotate.Phase.RUN,
   javax.safetycritical.annotate.Phase.CLEANUP})
@SCJMaySelfSuspend(false)
@SCJMayAllocate({})
public void scheduleNextReleaseTime(HighResolutionTime<?> time)
```

Changes the next scheduled release time for this handler. This method can take either an AbsoluteTime or a RelativeTime for its argument, and the handler will be released as if it was created using that type for its time parameter. An absolute time in the past is equivalent to a relative time of 0.0. The rescheduling value will be effective before the return of the method.

If there is no outstanding scheduled next release, this sets one.

If scheduleNextReleaseTime is invoked with a null parameter, any next release time is descheduled.

Throws IllegalArgumentException if time is a negative RelativeTime value or clock associated with time is not the same clock that was used during construction.

4.5.17 Class javax.safetycritical.AperiodicEventHandler

Declaration

```
@SCJAllowed(javax.safetycritical.annotate.Level.LEVEL_1)
 public abstract class AperiodicEventHandler extends
     javax.safetycritical.ManagedEventHandler
```

Description

This class encapsulates an aperiodic event handler. Concrete subclasses must implement the handleAsyncEvent method and may override the default cleanUp method.

There is no application access to the RTSJ fireCount mechanisms, so the associated methods are missing; see the AperiodicParameters class description for additional information.

Note that the values in parameters passed to the constructors are those that will be used by the infrastructure. Changing these values after construction will have no impact on the created event handler.

Constructors

```
@SCJAllowed(javax.safetycritical.annotate.Level.LEVEL_1)
@SCJPhase({javax.safetycritical.annotate.Phase.INITIALIZATION})
@SCJMaySelfSuspend(false)
@SCJMayAllocate({
    javax.safetycritical.annotate.AllocationContext.CURRENT,
    javax.safetycritical.annotate.AllocationContext.INNER,
    javax.safetycritical.annotate.AllocationContext.OUTER})
public AperiodicEventHandler(PriorityParameters priority,
  AperiodicParameters release,
  ScopeParameters storage,
  ConfigurationParameters config)
```

Constructs an aperiodic event handler that can be explicitly released.

The values passed as constructor parameters are captured at construction time. Any subsequent mutation of the parameter objects has no effect on the behavior of the constructed object.

priority — specifies the priority parameters for this handler. Must not be null.

release — specifies the release parameters for this handler. A null parameter indicates that there is no deadline associated with this handler.

storage — - it must not be null. specifies the ScopeParameters for this handler.

config — specifies the ConfigurationParameters for this handler.

Throws IllegalArgumentException if priority or storage is null; or when any deadline miss handler specified is not an AperiodicEventHandler.

Methods

```
@SCJAllowed
@SCJMaySelfSuspend(false)
@SCJMayAllocate({})
@SCJPhase({
    javax.safetycritical.annotate.Phase.STARTUP,
    javax.safetycritical.annotate.Phase.INITIALIZATION,
    javax.safetycritical.annotate.Phase.RUN,
    javax.safetycritical.annotate.Phase.CLEANUP })
public final void release( )
```

Release this aperiodic event handler.

4.5.18 Class javax.safetycritical.AperiodicLongEventHandler

Declaration

```
@SCJAllowed(javax.safetycritical.annotate.Level.LEVEL_1)
 public abstract class AperiodicLongEventHandler extends
     javax.safetycritical.ManagedLongEventHandler
```

Description

This class encapsulates an aperiodic event handler that is passed a long value when it is released. Concrete subclasses must implement the handleAsyncEvent method and may override the default cleanUp method.

There is no programmer access to the RTSJ fireCount mechanisms, so the associated methods are missing; see the AperiodicParameters class description for additional information.

Note that the values in parameters classes passed to the constructors are those that will be used by the infrastructure. Changing these values after construction will have no impact on the created event handler.

Constructors

```
@SCJAllowed(javax.safetycritical.annotate.Level.LEVEL_1)
@SCJPhase({javax.safetycritical.annotate.Phase.INITIALIZATION})
@SCJMaySelfSuspend(false)
@SCJMayAllocate({
   javax.safetycritical.annotate.AllocationContext.CURRENT,
   javax.safetycritical.annotate.AllocationContext.INNER,
   javax.safetycritical.annotate.AllocationContext.OUTER})
public AperiodicLongEventHandler(PriorityParameters priority,
  AperiodicParameters release,
  ScopeParameters storage,
  ConfigurationParameters config)
```

Constructs an aperiodic long event handler that can be released.

The values passed as constructor parameters are captured at construction time. Any subsequent mutation of the parameter objects has no effect on the behavior of the constructed object.

priority — specifies the priority parameters for this handler; it must not be null.

release — specifies the release parameters for this handler. A null parameter indicates that there is no deadline associated with this handler.

storage — specifies the ScopeParameters for this handler; it must not be null.

config — specifies the ConfigurationParameters for this handler.

Throws IllegalArgumentException if priority or storge is null; or when any deadline miss handler specified in release is not an AperiodicHandler.

Methods

```
@SCJAllowed
@SCJMaySelfSuspend(false)
@SCJMayAllocate({})
@SCJPhase({
   javax.safetycritical.annotate.Phase.STARTUP,
   javax.safetycritical.annotate.Phase.INITIALIZATION,
   javax.safetycritical.annotate.Phase.RUN,
   javax.safetycritical.annotate.Phase.CLEANUP })
public final void release(long data)
```

Release this long aperiodic event handler.

data — is the value associated with the release.

4.6 Threads and Real-Time Threads

In keeping with the approach outlined above for events and their handlers, the thread APIs are also significantly simplified relative to their counterparts in the RTSJ. They are shown in Figure 4.3.

Unless indicated otherwise, the classes defined in this section are thread safe.

4.6.1 Class java.lang.Thread

Declaration

```
@SCJAllowed
 public class Thread implements java.lang.Runnable extends java.lang.Object
```

Description

The Thread class is not directly available to the application in SCJ. However, some of its static methods are used, and the infrastructure will extend from this class and hence some of its methods are inherited.

Methods

```
@SCJAllowed(javax.safetycritical.annotate.Level.LEVEL_2)
@SCJPhase({
   javax.safetycritical.annotate.Phase.INITIALIZATION,
   javax.safetycritical.annotate.Phase.RUN,
   javax.safetycritical.annotate.Phase.CLEANUP})
@SCJMaySelfSuspend(false)
@SCJMayAllocate({})
public static java.lang.Thread.UncaughtExceptionHandler
     getDefaultUncaughtExceptionHandler( )
```

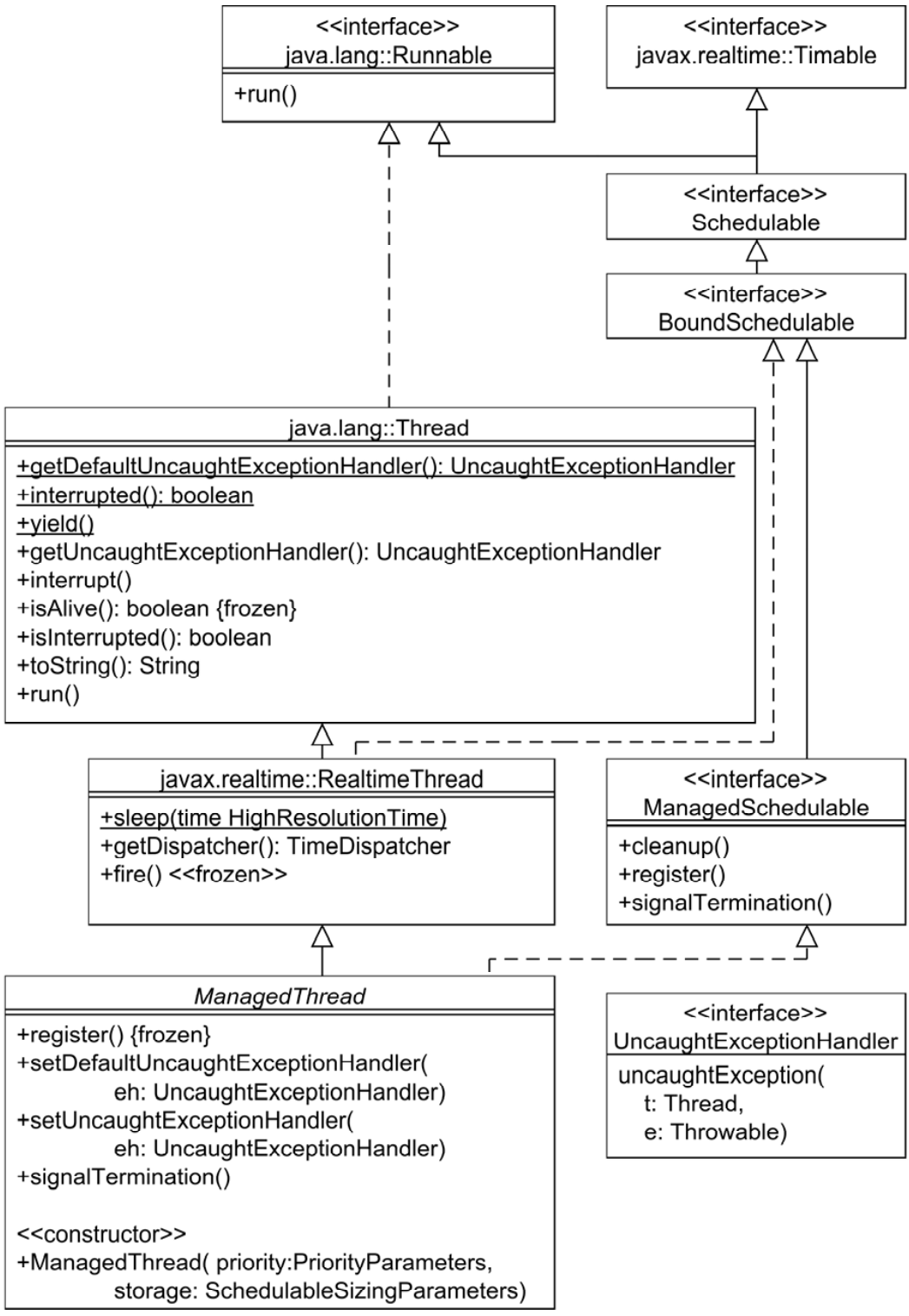

Figure 4.3: Thread classes

Gets the current thread's default uncaught exeption handler. Allocates no memory. Does not allow this to escape local variables. The result returned from this method may reside in scoped memory in some scope that encloses this.

returns the default handler for uncaught exceptions.

```
@SCJAllowed(javax.safetycritical.annotate.Level.LEVEL_2)
@SCJPhase({
   javax.safetycritical.annotate.Phase.INITIALIZATION,
   javax.safetycritical.annotate.Phase.RUN,
   javax.safetycritical.annotate.Phase.CLEANUP})
@SCJMaySelfSuspend(false)
@SCJMayAllocate({})
public java.lang.Thread.UncaughtExceptionHandler getUncaughtExceptionHandler( )
```

Get the thread's uncaught exception handler.

returns the handler invoked when this thread abruptly terminates due to an uncaught exception. Allocates no memory. Does not allow "this" to escape local variables. The result returned from this method may reside in scoped memory in some scope that encloses "this".

```
@SCJAllowed(javax.safetycritical.annotate.Level.LEVEL_2)
@SCJPhase({
   javax.safetycritical.annotate.Phase.INITIALIZATION,
   javax.safetycritical.annotate.Phase.RUN,
   javax.safetycritical.annotate.Phase.CLEANUP})
@SCJMaySelfSuspend(false)
@SCJMayAllocate({})
public void interrupt( )
```

Interrupts the thread. Allocates no memory. Does not allow this to escape local variables.

```
@SCJAllowed(javax.safetycritical.annotate.Level.LEVEL_2)
@SCJMaySelfSuspend(false)
@SCJPhase({
   javax.safetycritical.annotate.Phase.INITIALIZATION,
   javax.safetycritical.annotate.Phase.RUN,
   javax.safetycritical.annotate.Phase.CLEANUP})
@SCJMayAllocate({})
public static boolean interrupted( )
```

Tests whether the thread has been interrupted. The interrupted status of the thread is cleared by this method. Allocates no memory. Does not allow this to escape local variables.

returns true if the current thread has been interrupted.

```
@SCJAllowed(javax.safetycritical.annotate.Level.LEVEL_2)
@SCJPhase({
    javax.safetycritical.annotate.Phase.INITIALIZATION,
    javax.safetycritical.annotate.Phase.RUN,
    javax.safetycritical.annotate.Phase.CLEANUP})
@SCJMaySelfSuspend(false)
@SCJMayAllocate({})
public final boolean isAlive( )
```

Tests whether the thread is alive.

returns true if the current thread has not returned from run(). Allocates no memory. Does not allow this to escape local variables.

```
@SCJAllowed(javax.safetycritical.annotate.Level.LEVEL_2)
@SCJPhase({
    javax.safetycritical.annotate.Phase.INITIALIZATION,
    javax.safetycritical.annotate.Phase.RUN,
    javax.safetycritical.annotate.Phase.CLEANUP})
@SCJMaySelfSuspend(false)
@SCJMayAllocate({})
public boolean isInterrupted( )
```

Tests whether the thread has been interrupted. The interrupted status of the thread is not affected by this method. Allocates no memory. Does not allow this to escape local variables.

returns true if a thread has been interrupted.

```
@SCJAllowed(javax.safetycritical.annotate.Level.SUPPORT)
@SCJMayAllocate({
    javax.safetycritical.annotate.AllocationContext.CURRENT,
    javax.safetycritical.annotate.AllocationContext.INNER,
    javax.safetycritical.annotate.AllocationContext.OUTER})
@SCJMaySelfSuspend(true)
@SCJPhase({
    javax.safetycritical.annotate.Phase.STARTUP,
    javax.safetycritical.annotate.Phase.INITIALIZATION,
    javax.safetycritical.annotate.Phase.RUN,
    javax.safetycritical.annotate.Phase.CLEANUP })
public void run( )
```

This method is overridden by the application to do the work desired for this thread. This method should not be directly called by the application.

```
@SCJPhase({
    javax.safetycritical.annotate.Phase.INITIALIZATION,
    javax.safetycritical.annotate.Phase.RUN,
    javax.safetycritical.annotate.Phase.CLEANUP})
@SCJMaySelfSuspend(false)
@SCJMayAllocate({javax.safetycritical.annotate.AllocationContext.CURRENT})
@SCJAllowed
public java.lang.String toString( )
```

Gets the name and priority for the thread.

returns a string representation of this thread, including the thread's name and priority. Does not allow this to escape local variables. Allocates a String and associated internal "structure" (e.g. char[]) in caller's scope.

```
@SCJAllowed(javax.safetycritical.annotate.Level.LEVEL_2)
@SCJPhase({
    javax.safetycritical.annotate.Phase.INITIALIZATION,
    javax.safetycritical.annotate.Phase.RUN,
    javax.safetycritical.annotate.Phase.CLEANUP})
@SCJMaySelfSuspend(false)
@SCJMayAllocate({})
public static void yield( )
```

Causes the thread to yield to other threads that may be ready to run. Causes the currently executing thread object to temporary pause and allow other threads to execute.

4.6.2 Class javax.realtime.RealtimeThread

Declaration

```
@SCJAllowed
 public class RealtimeThread implements javax.realtime.BoundSchedulable,
     javax.realtime.AsyncTimable extends java.lang.Thread
```

Description

Real-time threads cannot be directly created by an SCJ application. However, they are needed by the infrastructure to support ManagedThreads.

The class declares some static methods that can be used by all managed schedulables. For example, the spin method can be used at Level 0, hence the class is visible at Level 0.

Methods

```
@SCJAllowed(javax.safetycritical.annotate.Level.SUPPORT)
@SCJMaySelfSuspend(false)
@SCJMayAllocate({})
@Override
@SCJPhase({
    javax.safetycritical.annotate.Phase.STARTUP,
    javax.safetycritical.annotate.Phase.INITIALIZATION,
    javax.safetycritical.annotate.Phase.RUN,
    javax.safetycritical.annotate.Phase.CLEANUP })
public final void fire( )
```

Used by the SCJ infrastructure to support the time release of real-time threads and timers with user-defined clocks. Should not be called by the application.

```
@SCJMaySelfSuspend(false)
@SCJPhase({
    javax.safetycritical.annotate.Phase.STARTUP,
    javax.safetycritical.annotate.Phase.INITIALIZATION,
    javax.safetycritical.annotate.Phase.RUN,
    javax.safetycritical.annotate.Phase.CLEANUP})
@SCJMayAllocate({})
@SCJAllowed(javax.safetycritical.annotate.Level.LEVEL_1)
@Override
public javax.realtime.Affinity getAffinity( )
```

```
@SCJMaySelfSuspend(false)
@SCJPhase({
    javax.safetycritical.annotate.Phase.STARTUP,
    javax.safetycritical.annotate.Phase.INITIALIZATION})
@SCJMayAllocate({})
@SCJAllowed(javax.safetycritical.annotate.Level.LEVEL_1)
@Override
public void setAffinity(Affinity set)
  throws java.lang.IllegalArgumentException,
      javax.realtime.ProcessorAffinityException, java.lang.NullPointerException
```

```
@SCJAllowed(javax.safetycritical.annotate.Level.LEVEL_2)
@SCJMaySelfSuspend(true)
@SCJMayAllocate({})
@SCJPhase({
    javax.safetycritical.annotate.Phase.INITIALIZATION,
    javax.safetycritical.annotate.Phase.RUN,
    javax.safetycritical.annotate.Phase.CLEANUP})
public static void sleep(HighResolutionTime<?> time)
  throws java.lang.InterruptedException
```

Removes the currently execution schedulable object from the set of runnable schedulable objects until time.

Throws InterruptedException when the thread is interrupted by interrupt() during the time between calling this method and returning from it. This exception cannot be thrown if the method is called from a managed event handler.

Throws IllegalArgumentException when time is null, when time is a relative time less than zero, or when the Chronograph of time is not a Clock.

```
@SCJAllowed
@SCJMaySelfSuspend(true)
@SCJMayAllocate({})
@SCJPhase({
    javax.safetycritical.annotate.Phase.STARTUP,
    javax.safetycritical.annotate.Phase.INITIALIZATION,
    javax.safetycritical.annotate.Phase.RUN,
    javax.safetycritical.annotate.Phase.CLEANUP })
public static void spin(int nanos)
  throws java.lang.InterruptedException,
     java.lang.ClassCastException, java.lang.IllegalArgumentException
```

The same as calling spin(HighResolutionTime) with a relative time on the default real-time clock,of zero milliseconds, and nanos.

nanos — the number of nanoseconds to wait.

Throws InterruptedException when the thread is interrupted by interrupt() during the time between calling this method and returning from it. This exception cannot be thrown if the method is called from a managed event handler.

Throws IllegalArgumentException when nanos is less than zero.

```
@SCJAllowed
@SCJMaySelfSuspend(false)
@SCJMayAllocate({})
@SCJPhase({
    javax.safetycritical.annotate.Phase.STARTUP,
    javax.safetycritical.annotate.Phase.INITIALIZATION,
    javax.safetycritical.annotate.Phase.RUN,
    javax.safetycritical.annotate.Phase.CLEANUP })
public static void spin(HighResolutionTime<?> time)
  throws java.lang.InterruptedException,
     java.lang.ClassCastException, java.lang.IllegalArgumentException
```

Similar to sleep(HighResolutionTime) except it performs a busy wait by polling the Chronograph for the duration of time.

time — an absolute or relative time at which to stop spinning.

Throws InterruptedException when the thread is interrupted by interrupt() during the time between calling this method and returning from it. This exception cannot be thrown if the method is called from a managed event handler.

Throws IllegalArgumentException when time is null , or when time is a relative time less than zero.

```
@SCJAllowed(javax.safetycritical.annotate.Level.LEVEL_2)
@SCJMaySelfSuspend(true)
@SCJMayAllocate({})
@SCJPhase({
   javax.safetycritical.annotate.Phase.INITIALIZATION,
   javax.safetycritical.annotate.Phase.RUN,
   javax.safetycritical.annotate.Phase.CLEANUP})
public static void suspend(HighResolutionTime<?> time)
```

The same as sleep(HighResolutionTime) except that it is not interruptible.

time — an absolute or relative time until which to suspend.

Throws IllegalArgumentException when time is null, when time is a relative time less than zero, or when the Chronograph of time is not a Clock.

4.6.3 Class javax.safetycritical.ManagedThread

Declaration

```
@SCJAllowed(javax.safetycritical.annotate.Level.LEVEL_2)
 public abstract class ManagedThread implements
     javax.safetycritical.ManagedSchedulable extends javax.realtime.RealtimeThread
```

Description

This class enables a mission to keep track of all the no-heap realtime threads that are created during the mission's initialization phase.

Note that the values in parameters classes passed to the constructors are those that will be used by the infrastructure. Changing these values after construction will have no impact on the created no-heap real-time thread. Managed threads have no release parameters.

Constructors

```
@SCJPhase({javax.safetycritical.annotate.Phase.INITIALIZATION})
@SCJMaySelfSuspend(false)
@SCJAllowed
@SCJMayAllocate({
   javax.safetycritical.annotate.AllocationContext.CURRENT,
```

```
    javax.safetycritical.annotate.AllocationContext.INNER,
    javax.safetycritical.annotate.AllocationContext.OUTER})
public ManagedThread(PriorityParameters priority,
  ReleaseParameters release,
  ScopeParameters scope,
  ConfigurationParameters storage,
  Runnable logic)
```

Creates a thread that is managed by the enclosing mission.

The priority represented by PriorityParameters is consulted only once, at construction time.

priority — specifies the priority parameters for this managed thread; it must not be null.

storage — specifies the memory parameters for this thread. May not be null.

logic — the code for this managed thread.

Throws IllegalArgumentException if priority or storage is null.

Methods

```
@SCJAllowed(javax.safetycritical.annotate.Level.SUPPORT)
@SCJPhase({javax.safetycritical.annotate.Phase.CLEANUP})
@SCJMaySelfSuspend(false)
@Override
@SCJMayAllocate({
    javax.safetycritical.annotate.AllocationContext.CURRENT,
    javax.safetycritical.annotate.AllocationContext.INNER,
    javax.safetycritical.annotate.AllocationContext.OUTER})
public void cleanUp( )
```

Runs any end-of-mission clean up code associated with this schedulable object.
see ManagedSchedulable.cleanUp()

```
@SCJPhase({javax.safetycritical.annotate.Phase.INITIALIZATION})
@SCJMaySelfSuspend(false)
@SCJMayAllocate({})
@Override
@SCJAllowed
public void register( )
```

Register this managed thread with the current mission.

see javax.safetycritical.ManagedSchedulable.register().

```
@SCJPhase({javax.safetycritical.annotate.Phase.INITIALIZATION})
@SCJMaySelfSuspend(false)
@SCJMayAllocate({})
@SCJAllowed(javax.safetycritical.annotate.Level.LEVEL_2)
public static void setDefaultUncaughtExceptionHandler(
  Thread.UncaughtExceptionHandler eh)
```

This method is used by the application to define an exception handler that will handle uncaught exceptions.

```
@SCJAllowed(javax.safetycritical.annotate.Level.LEVEL_2)
@SCJPhase({javax.safetycritical.annotate.Phase.INITIALIZATION})
@SCJMaySelfSuspend(false)
@SCJMayAllocate({})
public void setUncaughtExceptionHandler(Thread.UncaughtExceptionHandler eh)
```

Set the current uncaught exception handler.

eh — the UncaughtExceptionHandler to be set for this managed thread. The eh argument must reside in a scope that encloses the scope of this.

```
@SCJAllowed(javax.safetycritical.annotate.Level.SUPPORT)
@SCJMaySelfSuspend(false)
@SCJMayAllocate({})
@SCJPhase({
   javax.safetycritical.annotate.Phase.RUN,
   javax.safetycritical.annotate.Phase.CLEANUP})
public void signalTermination( )
```

Called by the infrastructure to indicate that the enclosing mission has been instructed to terminate. The default behaviour is no action.

4.7 Scheduling and Related Activities

Level 0 applications are scheduled by a cyclic executive where the schedule is created manually or by static analysis tools offline. Level 1 and Level 2 applications are assumed to be scheduled by a FIFO preemptive priority scheduler.

Unless indicated otherwise, the classes defined in this section are thread safe

4.7.1 Class javax.safetycritical.CyclicSchedule

See Section 3.4.5.

4.7.2 Class javax.safetycritical.CyclicExecutive

See Section 3.4.6.

4.7.3 Class javax.realtime.Scheduler

Declaration

```
@SCJAllowed
 public abstract class Scheduler extends java.lang.Object
```

Description

The RTSJ supported generic on-line feasibility analysis via the Scheduler class prior to RTSJ version 2.0, but this is now deprecated in version 2.0. SCJ supports only off-line schedulability analysis; hence all of the methods in this class are omitted.

Methods

```
@SCJAllowed
@SCJPhase({
   javax.safetycritical.annotate.Phase.INITIALIZATION,
   javax.safetycritical.annotate.Phase.RUN,
   javax.safetycritical.annotate.Phase.CLEANUP})
@SCJMaySelfSuspend(false)
@SCJMayAllocate({})
public static javax.realtime.Schedulable currentSchedulable( )
```

Gets the current schedulable.

returns a reference to the calling Schedulable.

Throws UnsupportedOperationException if called from an interrupt handler.

4.7.4 Class javax.realtime.PriorityScheduler

Declaration

```
@SCJAllowed
 public abstract class PriorityScheduler extends javax.realtime.Scheduler
```

Description

Priority-based dispatching is supported at Level 1 and Level 2. The only access to the priority scheduler is for obtaining the minimum and maximum priority.

Methods

```
@SCJAllowed
@SCJMaySelfSuspend(false)
@SCJMayAllocate({})
@SCJPhase({
    javax.safetycritical.annotate.Phase.STARTUP,
    javax.safetycritical.annotate.Phase.INITIALIZATION,
    javax.safetycritical.annotate.Phase.RUN,
    javax.safetycritical.annotate.Phase.CLEANUP })
public abstract int getMaxPriority( )
```

Gets the maximum software real-time priority supported by this scheduler.

returns the maximum priority supported by this scheduler.

```
@SCJAllowed
@SCJMaySelfSuspend(false)
@SCJMayAllocate({})
@SCJPhase({
    javax.safetycritical.annotate.Phase.STARTUP,
    javax.safetycritical.annotate.Phase.INITIALIZATION,
    javax.safetycritical.annotate.Phase.RUN,
    javax.safetycritical.annotate.Phase.CLEANUP })
public abstract int getMinPriority( )
```

Gets the minimum software real-time priority supported by this scheduler.

returns the minimum priority supported by this scheduler.

```
@SCJAllowed
@SCJMaySelfSuspend(false)
@SCJMayAllocate({})
@SCJPhase({
    javax.safetycritical.annotate.Phase.STARTUP,
    javax.safetycritical.annotate.Phase.INITIALIZATION,
    javax.safetycritical.annotate.Phase.RUN,
    javax.safetycritical.annotate.Phase.CLEANUP })
public abstract int getNormPriority( )
```

returns the normal software real-time priority supported by this scheduler.

4.7.5 Class javax.realtime.FirstInFirstOutScheduler

Declaration

```
@SCJAllowed(javax.safetycritical.annotate.Level.LEVEL_1)
 public class FirstInFirstOutScheduler extends javax.realtime.PriorityScheduler
```

Description

A version of **javax.realtime.PriorityScheduler** where once a thread is scheduled at a given priority, it runs until it is blocked or is preempted by a higher priority thread. When preempted, it remains the next thread ready for its priority. This is the default scheduler for realtime tasks. It represents the required (by the RTSJ) priority-based scheduler. The default instance is the base scheduler which does fixed priority, preemptive scheduling.

Methods

```
@SCJAllowed
@SCJMayAllocate({})
@SCJMaySelfSuspend(false)
@Override
@SCJPhase({
   javax.safetycritical.annotate.Phase.STARTUP,
   javax.safetycritical.annotate.Phase.INITIALIZATION,
   javax.safetycritical.annotate.Phase.RUN,
   javax.safetycritical.annotate.Phase.CLEANUP })
public int getMaxPriority( )
```

Obtain the maximum priority available for a schedulable managed by this scheduler.

returns The value of the maximum priority.

```
@SCJAllowed
@SCJMayAllocate({})
@SCJMaySelfSuspend(false)
@Override
@SCJPhase({
   javax.safetycritical.annotate.Phase.STARTUP,
   javax.safetycritical.annotate.Phase.INITIALIZATION,
   javax.safetycritical.annotate.Phase.RUN,
   javax.safetycritical.annotate.Phase.CLEANUP })
public int getMinPriority( )
```

Obtain the minimum priority available for a schedulable managed by this scheduler.

returns The minimum priority used by this scheduler.

```
@SCJAllowed
@SCJMayAllocate({})
@SCJMaySelfSuspend(false)
@Override
@SCJPhase({
   javax.safetycritical.annotate.Phase.STARTUP,
```

```
    javax.safetycritical.annotate.Phase.INITIALIZATION,
    javax.safetycritical.annotate.Phase.RUN,
    javax.safetycritical.annotate.Phase.CLEANUP })
public int getNormPriority( )
```

Obtain the normal priority available for a schedulable managed by this scheduler.

returns The value of the normal priority.

```
@SCJAllowed(javax.safetycritical.annotate.Level.LEVEL_1)
@SCJMayAllocate({})
@SCJMaySelfSuspend(false)
@SCJPhase({
    javax.safetycritical.annotate.Phase.STARTUP,
    javax.safetycritical.annotate.Phase.INITIALIZATION,
    javax.safetycritical.annotate.Phase.RUN,
    javax.safetycritical.annotate.Phase.CLEANUP })
public static javax.realtime.FirstInFirstOutScheduler instance( )
```

Obtain a reference to the distinguished instance of {@code PriorityScheduler} which is the system's base scheduler.

returns A reference to the distinguished instance {@code PriorityScheduler}.

4.7.6 Class javax.realtime.Affinity

Declaration

```
@SCJAllowed
 public final class Affinity extends java.lang.Object
```

Description

This class is the API for all processor-affinity-related aspects of the RTSJ that are relevant to the SCJ. It includes a factory that generates Affinity objects. The explicit setting of the affinity of SCJ managed schedulables is performed during its mission initialisation phase when the managed schedulable is registered. If no affinity is set, the managed schedulable inherits the affinity of its mission sequencer.

An affinity set is a set of processors that can be associated with a real-time thread or async event handler. For SCJ, an affinity set is associated to a managed schedulable. Each implementation supports an array of predefined affinity sets. They can be used either to reflect the scheduling arrangement of the underlying OS or they can be used by the system designer to impose defaults for,

schedulable objects. An application is only allowed to dynamically create new affinity sets with cardinality of one. This restriction reflects the concern that not all operating systems will support multiprocessor affinity sets.

The processor membership of an affinity set is immutable. The schedulable object associations to an affinity set are mutable.

The internal representation of an affinity set in an Affinity instance is not specified, but the representation that is used to communicate with this class is a BitSet where each bit corresponds to a logical processor ID. The relationship between logical and physical processors is implementation defined, and may differ from one implementation to another.

The affinity set factory may be used to create affinity sets with a single processor member at any time, though this operation only supports processor members that are valid as the processor affinity for a schedulable object (at the time of the affinity set's creation.) The factory cannot create an affinity set with more than one processor member, but such affinity sets are supported. They may be internally created by the SCJ infrastructure, probably at start up time.

The set of affinity sets created by the infrastructure at start up (the predefined set) is visible through the getPredefinedAffinities(Affinity[]) method. In SCJ the initial mission sequencer has an affinity equal to getPredefinedAffinities(Affinity[])[0]; that is the first element of the returned array

External changes to the set of processors available to the SCJ infrastructure is likely to cause serious trouble ranging from violation of assumptions underlying schedulability analysis to freezing the entire SCJ program, so if a system is capable of such manipulation it should not exercise it on SCJ processes.

There is no public constructor for this class. All instances must be generated by the factory method (generate).

Methods

```
@SCJAllowed(javax.safetycritical.annotate.Level.LEVEL_2)
@SCJMaySelfSuspend(false)
@SCJMayAllocate({javax.safetycritical.annotate.AllocationContext.CURRENT})
@SCJPhase({
   javax.safetycritical.annotate.Phase.STARTUP,
   javax.safetycritical.annotate.Phase.INITIALIZATION,
   javax.safetycritical.annotate.Phase.RUN,
   javax.safetycritical.annotate.Phase.CLEANUP })
public static final javax.realtime.Affinity generate(BitSet bitSet)
```

Returns an Affinity set with the affinity bitSet and no associations.

Platforms that support specific affinity sets will register those Affinity instances with Affinity. They appear in the arrays returned bygetPredefinedAffinities() and getPredefinedAffinities(Affinity[]).

bitSet — The set of processors associated with the generated Affinity.

returns The resulting Affinity.

Throws NullPointerException when bitSet is null.

Throws IllegalArgumentException when bitSet does not refer to a valid set of processors, where "valid" is defined as the bitset from a pre-defined affinity set, or a bitset of cardinality one containing a processor from the set returned by getAvailableProcessors(). The definition of "valid set of processors" is system dependent; however, every set consisting of one valid processor makes up a valid bit set, and every bit set corresponding to a pre-defined affinity set is valid.

```
@SCJAllowed(javax.safetycritical.annotate.Level.LEVEL_1)
@SCJMaySelfSuspend(false)
@SCJMayAllocate({javax.safetycritical.annotate.AllocationContext.CURRENT})
@SCJPhase({
   javax.safetycritical.annotate.Phase.STARTUP,
   javax.safetycritical.annotate.Phase.INITIALIZATION,
   javax.safetycritical.annotate.Phase.RUN,
   javax.safetycritical.annotate.Phase.CLEANUP })
public static final java.util.BitSet getAvailableProcessors(BitSet dest)
```

In systems where the set of processors available to a process is dynamic (e.g., because of system management operations or because of fault tolerance capabilities), the result of this operation shall reflect the processsors that are currently allocated to the SCJ infrastructure and are currently available to execute tasks.

dest — If dest is non-null, use dest for the returned value. If it is null, create a new BitSet.

returns the set of processors representing the set of processors currently valid for the bitset argument to generate(BitSet).

```
@SCJAllowed(javax.safetycritical.annotate.Level.LEVEL_1)
@SCJMaySelfSuspend(false)
@SCJMayAllocate({javax.safetycritical.annotate.AllocationContext.CURRENT})
@SCJPhase({
   javax.safetycritical.annotate.Phase.STARTUP,
   javax.safetycritical.annotate.Phase.INITIALIZATION,
   javax.safetycritical.annotate.Phase.RUN,
   javax.safetycritical.annotate.Phase.CLEANUP })
public static final java.util.BitSet getAvailableProcessors( )
```

This method is equivalent to getAvailableProcessors(null).

returns the set of processors available to the application.

```
@SCJAllowed(javax.safetycritical.annotate.Level.LEVEL_1)
@SCJMaySelfSuspend(false)
@SCJMayAllocate({javax.safetycritical.annotate.AllocationContext.CURRENT})
@SCJPhase({
    javax.safetycritical.annotate.Phase.STARTUP,
    javax.safetycritical.annotate.Phase.INITIALIZATION,
    javax.safetycritical.annotate.Phase.RUN,
    javax.safetycritical.annotate.Phase.CLEANUP })
public static final javax.realtime.Affinity[] getPredefinedAffinities( )
```

Equivalent to invoking getPredefinedAffinitySets(null).

returns an array of the pre-defined affinity sets.

```
@SCJAllowed(javax.safetycritical.annotate.Level.LEVEL_1)
@SCJMaySelfSuspend(false)
@SCJMayAllocate({javax.safetycritical.annotate.AllocationContext.CURRENT})
@SCJPhase({
    javax.safetycritical.annotate.Phase.STARTUP,
    javax.safetycritical.annotate.Phase.INITIALIZATION,
    javax.safetycritical.annotate.Phase.RUN,
    javax.safetycritical.annotate.Phase.CLEANUP })
public static final javax.realtime.Affinity[] getPredefinedAffinities(
  Affinity [] dest)
```

Return an array containing all affinity sets that were predefined by the infrastructure.

dest — The destination array, or null.

returns dest or a newly created array if dest was null, populated with references to the pre-defined affinity sets.

If dest has excess entries, they are filled with null.

Throws IllegalArgumentException when dest is not large enough.

```
@SCJAllowed(javax.safetycritical.annotate.Level.LEVEL_1)
@SCJMaySelfSuspend(false)
@SCJMayAllocate({})
@SCJPhase({
    javax.safetycritical.annotate.Phase.STARTUP,
    javax.safetycritical.annotate.Phase.INITIALIZATION,
    javax.safetycritical.annotate.Phase.RUN,
    javax.safetycritical.annotate.Phase.CLEANUP })
public static final int getPredefinedAffinitiesCount( )
```

Return the minimum array size required to store references to all the predefined processor affinity sets.

returns the minimum array size required to store references to all the predefined affinity sets.

```
@SCJAllowed(javax.safetycritical.annotate.Level.LEVEL_1)
@SCJMaySelfSuspend(false)
@SCJMayAllocate({javax.safetycritical.annotate.AllocationContext.CURRENT})
@SCJPhase({
    javax.safetycritical.annotate.Phase.STARTUP,
    javax.safetycritical.annotate.Phase.INITIALIZATION,
    javax.safetycritical.annotate.Phase.RUN,
    javax.safetycritical.annotate.Phase.CLEANUP })
public final java.util.BitSet getProcessors(BitSet dest)
```

Return a BitSet representing the processor affinity set of this Affinity.

dest — Set dest to the BitSet value. If dest is null, create a new BitSet in the current allocation context.

returns a BitSet representing the processor affinity set of this Affinity.

```
@SCJAllowed(javax.safetycritical.annotate.Level.LEVEL_1)
@SCJMaySelfSuspend(false)
@SCJMayAllocate({javax.safetycritical.annotate.AllocationContext.CURRENT})
@SCJPhase({
    javax.safetycritical.annotate.Phase.STARTUP,
    javax.safetycritical.annotate.Phase.INITIALIZATION,
    javax.safetycritical.annotate.Phase.RUN,
    javax.safetycritical.annotate.Phase.CLEANUP })
public final java.util.BitSet getProcessors( )
```

Return a BitSet representing the processor affinity set for this Affinity.

returns a newly created BitSet representing this Affinity.

```
@SCJAllowed(javax.safetycritical.annotate.Level.LEVEL_1)
@SCJMaySelfSuspend(false)
@SCJMayAllocate({})
@SCJPhase({
    javax.safetycritical.annotate.Phase.STARTUP,
    javax.safetycritical.annotate.Phase.INITIALIZATION,
    javax.safetycritical.annotate.Phase.RUN,
    javax.safetycritical.annotate.Phase.CLEANUP })
public final boolean isProcessorInSet(int processorNumber)
```

Ask whether a processor is included in this affinity set.

processorNumber — is a logical processor number

returns true if and only if processorNumber is a member of this affinity set.

4.7.7 Class jaxax.safetycritical.Services

Declaration

```
@SCJAllowed
 public class Services extends java.lang.Object
```

Description

This class provides a collection of static helper methods.

Methods

```
@SCJAllowed
@SCJMaySelfSuspend(false)
@SCJMayAllocate({})
@SCJPhase({
   javax.safetycritical.annotate.Phase.STARTUP,
   javax.safetycritical.annotate.Phase.INITIALIZATION,
   javax.safetycritical.annotate.Phase.RUN,
   javax.safetycritical.annotate.Phase.CLEANUP })
public static javax.safetycritical.annotate.Level getComplianceLevel( )
```

Get the current compliance level of the SCJ implementation.

returns the compliance level

```
@SCJAllowed(javax.safetycritical.annotate.Level.LEVEL_1)
@SCJMaySelfSuspend(false)
@SCJMayAllocate({})
@SCJPhase({
   javax.safetycritical.annotate.Phase.STARTUP,
   javax.safetycritical.annotate.Phase.INITIALIZATION,
   javax.safetycritical.annotate.Phase.RUN,
   javax.safetycritical.annotate.Phase.CLEANUP })
public static int getDefaultCeiling( )
```

Get the default ceiling priority for objects. By default, it is PriorityScheduler.getMaxPriority. It is assumed that this can be changed using an implementation configuration option.

returns the default ceiling priority.

```
@SCJAllowed(javax.safetycritical.annotate.Level.LEVEL_1)
@SCJPhase({javax.safetycritical.annotate.Phase.INITIALIZATION})
@SCJMaySelfSuspend(false)
@SCJMayAllocate({})
public static void setCeiling(Object obj, int pri)
```

Sets the ceiling priority of object obj The priority pri can be in the software or hardware priority range. Ceiling priorities are immutable.

obj — the object who ceiling is to be set.

pri — the ceiling value.

Throws IllegalSchedulableStateException if called outside the initialization phase of a mission.

4.8 Rationale for the SCJ Concurrency Model

Traditionally, most safety-critical systems were small and sequential, relying on cyclic executive scheduling to manually interleave the execution of any activities within time constraints. Demonstration that timeliness requirements have been met has been through construction and testing. The limitations of this approach are well known[5].

As safety-critical systems have become larger and more complex, there has been a gradual migration to programming models that support simple concurrent activities (threads, tasks, event handlers, etc.) that share an address space with each other. Whereas testing may have been adequate to prove reliable operations of sequential programs, it is not sufficient to demonstrate that timing constraints are met in a concurrent program. This is because of the large number of computational states possible in a concurrent program.

The transition from sequential to concurrent safety-critical systems has been accompanied by a shift from *deterministic* scheduling to *predictable* scheduling. Verification of timing requirements relies on schedulability analysis (called "feasibility analysis" in the RTSJ). Many of these techniques are now mature for single processor systems, with a firm mathematical foundation, and are accepted by many certification authorities (e.g., simple utilization-based or response-time analysis using rate-monotonic or deadline-monotonic priority ordering of threads). They rely on the ability to determine the worst-case execution time of threads and the amount of time they are blocked when accessing resources. The techniques for schedulability analysis, worst-case execution time analysis and blocking time analysis are beyond the scope of this specification. However, it is expected that schedulability analysis will be performed by most, if not all, safety-critical systems implementers, and their results will be included as evidence in any certification process for applications written according to this specification.

Specifying subsets of languages for use in safety-critical systems is accepted practice, as is constraining the way the subset is used. The Ada programming language, for example, has led the way in using concurrent activities (which it refers to as *tasks*) for

real-time, embedded programs, and as of the 2005 version of the language standard includes an explicit subset of Ada tasking constructs called the *Ravenscar Profile* that are amenable to formal certification against standards such as DO-178C.

The SCJ concurrency model aims to ease the migration from sequential to concurrent safety-critical systems. Level 0 is effectively a static cyclic scheduler, whereas Level 1 and Level 2 offer more dynamic, flexible scheduling.

4.8.1 Scheduling and Synchronization Issues

For schedulability analysis, all non-periodic activities must have bounded minimum interarrival times. In the RTSJ, the use of sporadic release parameters provides a mechanism with which the implementation can enforce these minimum arrival times. However, the SCJ specification does not provide for enforcing minimum inter-arrival times. Therefore, the SCJ specification uses the aperiodic parameter class and does not support sporadic release parameters, leaving the enforcement of minimum inter-arrival times to the application designer.

The priority ceiling emulation (PCE) protocol for bounding thread blocking during synchronized methods was originally optional in the RTSJ because many real-time operating systems support only priority inheritance. However, the priority ceiling protocol has emerged in recent years as a preferred approach on a single processor (under the assumption that schedulable objects do not self-suspend while holding a lock) because it has an efficient implementation and under some conditions, has the potential to guarantee that the program is deadlock free. It also ensures that a schedulable object is blocked at most once in a single release (at the start of its execution request).

Unlike the RTSJ, SCJ supports only the priority ceiling emulation protocol. Consequently, SCJ defines its own interface. It simply provides a static method in the javax.safetycritical.Services class that permits the ceiling of an object to be set.

The application of the priority ceiling emulation protocol to Java synchronized methods is not straightforward. Java allows lock retaining self-suspending operations such as, for example, the sleep and join methods when called from synchronized code. Furthermore, nested synchronized method calls that invoke the `Object.wait` method can release only one of the locks being held. For these reasons, the SCJ does not permit self-suspension while holding a lock at any compliance Level. At Level 2 where the use of the wait method is allowed; the following approaches are possible:

1. Prohibit all nested synchronized method calls. This seems draconian.

2. (Approach chosen for SCJ) Prohibit the call of the wait method from nested synchronized methods. This would probably be difficult to test statically and

would require a run-time exception to be raised (presumably IllegalMonitorStateException).

3. Allow all nested synchronized method calls with the standard Java semantics. On a single processor system, the PCE protocol would have to degrade to priority inheritance in this case (unfortunately then multiple possible blocking and the potential for deadlock). For multiprocessor systems, spinning for a lock would no longer be bounded.

4. Allow all nested synchronized method calls, but provide an annotation to indicate when synchronized code is suspension free.

SCJ prohibits the use of the wait() method in nested monitor calls and requires that methods indicate via annotations when they do not self-suspend. This means that a synchronized method can only call methods that will not self-suspend. Analysis tools can then check this condition and avoid the need for runtime checks.

It is expected that a call to a synchronized method will always behave as if the priority is raised during the method execution, even if there is no sharing of the object during execution.

Finally, SCJ does not support the Java synchronized statement. The reason for this is to simplify the static analysis techniques that are needed to determine which schedulable objects use which locks. This is required to determine ceiling priorities.

4.8.2 Multiprocessors

Although the techniques for analyzing the timing properties of multiprocessor systems are still relatively in their infancy, there is general agreement on the growing importance of multicore platforms for real-time and embedded systems, including safety-critical systems. For this reason, this specification provides support for programming multiprocessor platforms.

On a single processor, the priority ceiling emulation protocol has the following properties *if a schedulable object does not self-suspend while holding a lock*:

- no deadlocks can occur from the use of Java monitors, and
- each schedulable object can be blocked at most once during its release as a result of sharing two or more Java monitors with other schedulable objects.

The ceiling of each shared object must be at least the maximum priority of all the schedulable objects that access that shared object.

On a multiprocessor system (including multicore systems), the above properties still hold as long as Java monitors are not shared between schedulable objects executing on separate processors.

If schedulable objects on separate processors are sharing objects and they do not self-suspend while holding the monitor lock, then blocking can be bounded but the absence of deadlock cannot be assured by the PCE protocol alone.

The usual approach to waiting for a lock that is held by a schedulable object on a different processor is to spin (busy-wait). There are different approaches that can be used by an implementation such as, for example, maintaining a FIFO/Priority queue of spinning processors, and ensuring that the processors spin non-preemptively. SCJ does not mandate any particular approach but requires an implementation to document its approach (i.e., implementation-defined).

To avoid unbounded priority inversion, it is necessary to carefully set the ceiling values.

On a Level 1 system, the schedulable objects are fully partitioned among the processors using the scheduling allocation domain concept. The ceiling of every synchronized object that is accessible by more than one processor has to be set so that its synchronized methods execute in a non-preemptive manner. This is because there is no relationship between the priorities in one allocation domain and those in another.

On a Level 2 system, within a scheduling allocation domain, the value of the ceiling priorities must be higher than all the schedulable objects on all the processors in that scheduling allocation domain that can access the shared object. For monitors shared between scheduling allocation domains, the monitor methods must run in a non-preemptive manner.

Nested calls of synchronized methods, where the inner call blocks by calling the wait method, results in the outer lock being held throughout the wait period. In multiprocessor systems, this should generally be avoided if spinning is used for lock acquisition.

A lock is always required; using the priority model for locking is not sustainable with multiprocessors.

4.8.3 Schedulability Analysis and MultiProcessors

While schedulability analysis (also sometimes called feasibility analysis) techniques are mature for single processor systems, they are less mature for multiprocessor systems. Consequently, SCJ takes a very conservative approach. SCJ introduces the notion of a *scheduling allocation domain*.

At Level 0, a single processor scheduling allocation domain is supported that is implemented as a cyclic scheduler.

At Level 1, each scheduling allocation domain is a single processor and each processor is scheduled using fixed priority preemptive scheduling. The schedulability

analysis is equivalent to the well-known single processor schedulability analysis, but would be carried out for each scheduling allocation domain. Of course, the calculation of the blocking times will be different than they would be on a single processor system due to the potential for remote blocking.

At Level 2, each scheduling allocation domain may be more than one processor. Schedulable objects are globally scheduled according to fixed priority preemptive scheduling. The schedulability analysis for these systems is emerging and expected to mature over the next few years.

In all cases, the implementation-predefined affinity sets of the RTSJ are the scheduling allocation domains. Only Level 2 allows a new affinity set to be created. This is used to enable a schedulable object to fix its execution to a single processor. There are several reasons why this might be needed. These include: the schedulable objects use a device attached to a specific processor, or the schedulable object is CPU intensive, and to improve global utilization it needs to be run only on that processor, but can also share that CPU with other globally scheduled objects.

4.8.4 Impact of Clock Granularity

All time-triggered computation can suffer from release jitter. This is defined to be the variation in the actual time the computation becomes available for execution relative to its scheduled release time. The amount of release jitter depends on two factors. The first is the granularity of the clock/timer used to trigger the release. For example, a periodic event handler that is due to be released at absolute time T will actually be released at time $T + \Delta$. Δ is the difference between T and the first time the timer clock advances to T', where $T' \geq T$. The upper bound of Δ is the value returned from calling the getDrivePrecision method of the associated clock. It is for this reason that the implementation of release times for periodic activities must use absolute rather than relative time values, in order to avoid accumulating the drift.

The second contribution to release jitter is also related to the clock/timer. It is the time interval between T' being signaled by the clock/timer and the time this event is noticed by the underlying operating system or platform (e.g., perhaps because interrupts have been disabled). Figure 4.4 taken from [1] illustrates the delays that can occur.

4.8.5 Deadline Miss Detection

Although SCJ supports deadline miss detection, it is important to understand the intrinsic limitations of the facility. The SCJ supports deadline miss detection only for event handlers at levels 1 and 2. As explained in Section 4.8.4, all time-triggered

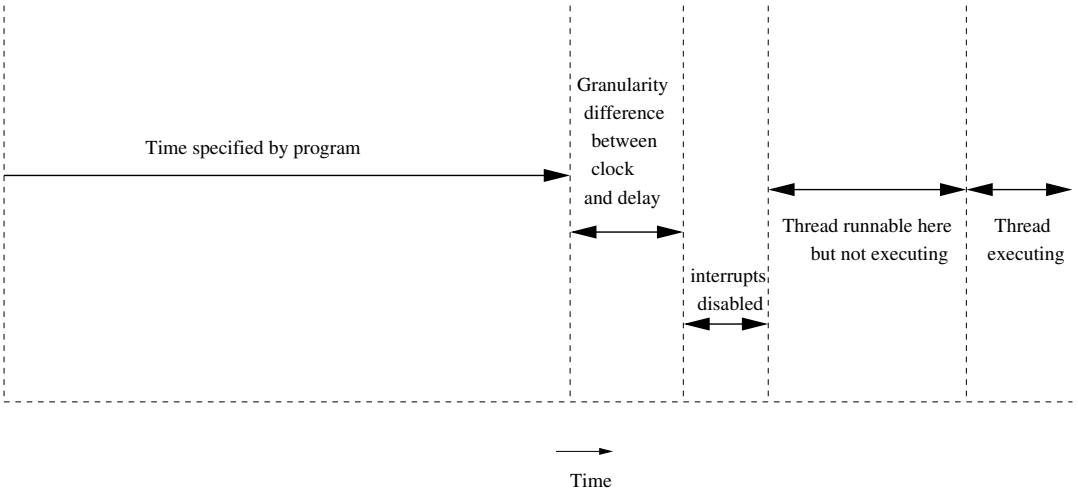

Figure 4.4: Granularity of delays

computations can suffer from release jitter, and this may result in a shifting of the time from which the time span representing the deadline is measured. If a deadline is not an integral multiple of the clock's tick size, the infrastructure must round the requested deadline up to the nearest multiple of the tick size. For example, if the clock tick is 10 microseconds, it is not possible to detect a deadline miss when the deadline is, say, 1 microsecond. Hence, an event handler whose release coincides with a clock tick at absolute time T, which has a deadline of $T + 1$ microsecond, will not have its deadline miss handler release until $T + 10$. It further follows, that a deadline can be missed but not detected. Consider an absolute deadline of D. Suppose that the next absolute time that the timer can recognize is $D + \Delta$. If the associated thread finishes after D but before $D + \Delta$, it will have missed its deadline, but this miss will have been undetected. Using the above example, the event handler can complete anytime before $T + 10$ and be deemed to meet its deadline.

Another limitation is due to the inherent race condition that is present when checking for deadline misses. A deadline miss is defined to occur if a managed event handler has not completed the computation associated with its release before its deadline. This completion event is signalled in the application code by return from the handleAsyncEvent method. When this method returns, the infrastructure cancels the timing event that signals the miss of a deadline. This is clearly a race condition. The timer event handler could be released to execute between the last executable statement within the handleAsyncEvent method and the canceling of the timer event. Hence a deadline miss could occasionally be signalled when arguably the application had performed all of its computation.

4.9 Compatibility

The following incompatibilities exist with the RTSJ:

- PCE is the default monitor control policy in SCJ whereas priority inheritance is the default in the RTSJ

Chapter 5

Interaction with Devices and External Events

5.1 Overview

Interactions between application programs and their environments can take several forms. They can be via

1. device interfaces and interrupt handling
2. operating system signals or other asynchronous notification mechanisms, and
3. high-level input and output capabilities such as files, sockets or some other connection-oriented mechanism.

This chapter describes the first two items of this list; namely the facilities that support interaction with input and output devices and interrupt handling, followed by the SCJ requirements for the specialized managed event handlers that are available for handling operating system signals. Chapter 6 discusses the SCJ support for the third item on this list.

The Overview and Rationale sections are not normative but are provided to improve understanding of the normative sections. All of the other sections of this chapter are normative.

5.2 Interaction with Input and Output Devices

In general, interfacing to input and output devices requires the programmer to be able to access the devices' control, status and data transfer registers, and to be able

to handle interrupts. The former is achieved by allowing the programmer to have controlled access to the physical device registers using a subset of the RTSJ raw memory access facilities. Optionally, SCJ supports first level interrupt handling when this can be provided by the underlying execution environment. These optional features are defined in the InterruptServiceRoutine class.

5.2.1 Semantics and Requirements

The RTSJ standardizes two means of accessing memory with specific properties: *physical memory* and *raw memory*. Physical memory provides a way of ensuring that specific objects get specific properties tied to particular areas of physical memory (e.g. non-cached memory areas). Raw Memory provides a means for accessing particular physical memory addresses as variables of Java's primitive data types, and thereby allows the application direct access to, for example, memory-mapped I/O registers. Java objects or their references cannot be stored in raw memory.

SCJ restricts the RTSJ API by not requiring any of the classes related to *physical memory*. The following specifies the SCJ's facilities for raw memory access:

- Each type of raw memory access is identified by a tagging class called RawMemoryRegion
  - The raw memory region MEMORY_MAPPED_REGION facilitates access to memory locations that are outside the main memory used by the JVM. It is used to to access input and output device registers when such registers are memory mapped.
  - The raw memory region IO_PORT_MAPPED_REGION facilitates access to locations that are outside the main memory used by the JVM. It is used to access input and output device registers when such registers are port-based and can only be accessed by special hardware instructions.
  - The application developer can define and register additional regions to support things like emulated access to devices or access to a bus over a bus controller.
- Access to raw memory is policed by implementation-defined objects, called **accessor objects**. These implement specification-defined interfaces (e.g., RawByte) and are created by implementation-defined factory objects.
- The RawMemoryRegionFactory interface defines the interface that all factories must support for creating accessor objects.
- Only Java integral types are supported.
- The RawMemoryFactory class defines the application programmer's interface to the raw memory facilities.

An overview of the supported classes and interfaces is shown in Figure: 5.1. Figure: 5.2 illustrates how they may be used.

1. Typically the SCJ infrastructure will create a factory to allow access to the raw memory areas it supports. As shown in 5.2, it creates a class for memory mapped IO.
2. The created factory is then registered with the raw memory factory manager.
3. The manager gets the name from the factory and checks that no factory has already been registered with that name.
4. The application, during one of the mission phases, is then able to request (from the raw memory factory manager) access to a particular type of raw memory.
5. The manager finds the appropriate factory and requests that it create an accessor object.
6. This object is then returned to the mission.

SCJ supports interaction with the external world by providing mechanism for interrupt handling and POSIX signal handling. The SCJ distinguishes between first-level support mechanisms and second-level support mechanisms. A first-level support mechanism implements the minimum platform-specific handling of the interrupt or operating system signal. In response to an interrupt or signal, there is a processor context switch, and the code for the first-level support mechanism is executed. A second-level support mechanism completes longer interrupt or signal processing tasks usually in the context of a managed event handler.

For operating system signals targeted at an SCJ program, the first-level support is provided by the SCJ infrastructure. The second-level support is provided by the program in the form of specialized SCJ managed event handlers (see Section 5.3). For device interrupts, SCJ optionally allows the program to provide first-level interrupt service routines as well as second-level application-defined managed event handlers (or managed threads). One exception is that the first-level interrupt service routine for the real-time clock is provided by the infrastructure. It is implementation-defined whether one or more additional device interrupts must be handled by an infrastructure-provided first-level interrupt service routine rather than an application-provided first-level interrupt service routine.

The primary goal of the first-level support code is to handle any immediate interaction with the external environment. For interrupt handling, this might include the storing of platform-specific information that is only available at the time of the interrupt and masking interrupts from the same source. Where appropriate, it may be necessary to establish an environment in which a second-level Java event handler can

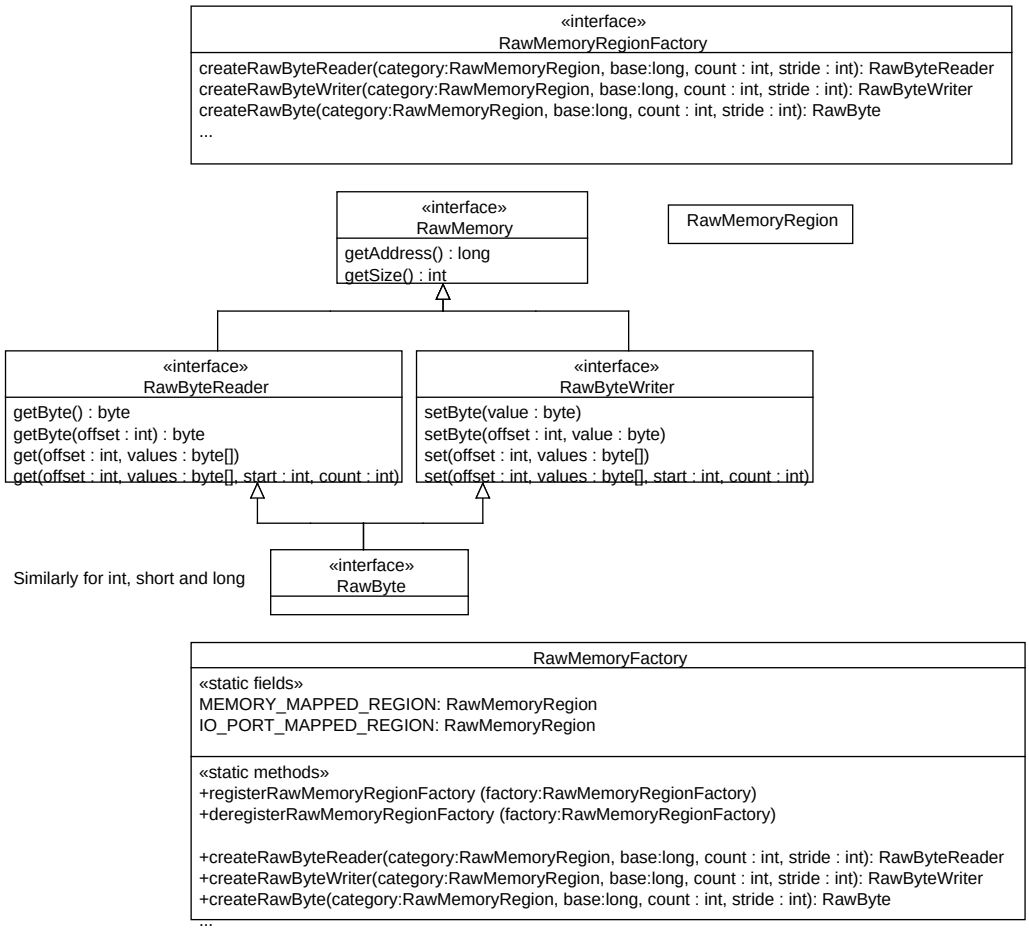

Figure 5.1: Raw memory classes and interfaces

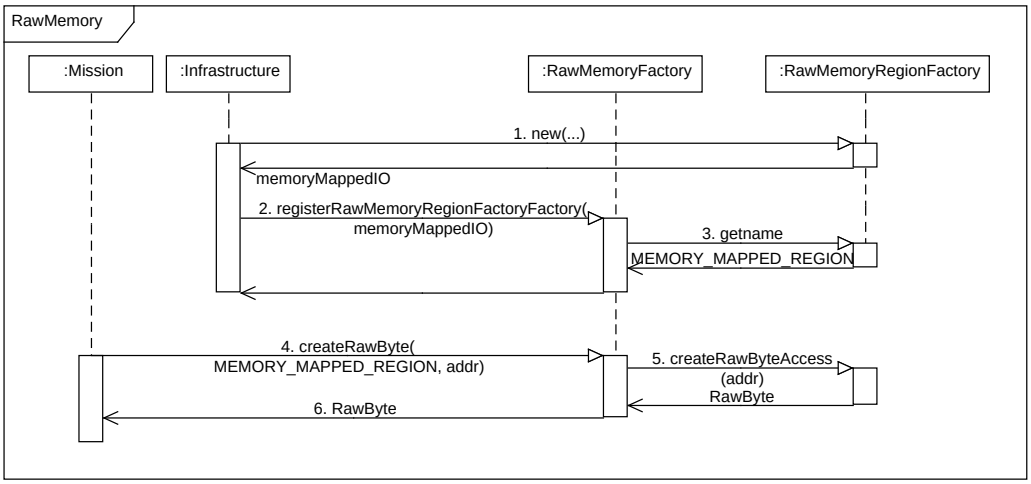

Figure 5.2: Raw memory classes interactions

be released. Similarly, for signal handling, the goal is to record the information associated with the signal and to establish an environment in which a second-level signal handler can be released. Typically, first-level support code executes at a high priority (often a hardware interrupt priority). It is, therefore, essential to keep this code to a minimum. One particular concern is that, for large systems, a significant number of event handlers may need to be released by the first-level support. For example, consider the first-level support for the real-time clock that, when a critical instance occurs, must release all waiting event handlers. This code can cause priority inversion as it may be releasing many low-priority handlers at an interrupt priority thereby delaying the execution of high priority handlers.

Like the RTSJ, SCJ fully defines its underlying model of interrupts. The following semantic model shall be supported by SCJ:

- An *occurrence* of an interrupt consists of its *generation* and *delivery*.
- Generation of the interrupt is the mechanism in the underlying hardware or system that makes the interrupt available to the Java infrastructure.
- Delivery is the action that invokes a *registered* interrupt service routine (ISR) in response to the generation of the interrupt. This may be performed by the JVM or application native code linked with the JVM, or directly by the hardware interrupt mechanism.
- Between generation and delivery, the interrupt is *pending*.
- Some or all interrupt generations may be inhibited. While an interrupt generation is inhibited, all deliveries of that interrupt shall be prevented. Whether such occurrences remain pending or are lost is implementation defined, but it is expected that the implementation shall make a best effort to avoid losing

pending interrupts.

- Certain implementation-defined interrupts are *reserved*. Reserved interrupts are either interrupts for which user-defined ISRs are not supported, or those that already have registered ISRs by some other implementation-defined means. For example, a clock interrupt, which is used for internal timekeeping by the infrastructure, is a reserved interrupt.
- An application-defined ISR can be registered to one or more non-reserved interrupts. Registering an ISR for an interrupt shall implicitly deregister any already registered ISR for that interrupt.
- While an ISR is registered to an interrupt, the handle method shall be called *once* for each delivery of that interrupt. The handle method should be synchronized. While the handle method executes, the corresponding interrupt (and all lower priority interrupts) shall be inhibited. The default allocation context of the handle method is a private implementation-provided memory area.
- The registration of an ISR shall be performed only during the initialization phase of a mission. Any ISR registered during the initialization phase of a mission shall be automatically deregistered by the infrastructure when the mission completes.
- An exception propagated from the handle method shall be caught by the SCJ infrastructure and result in the uncaughtException method being called in the associated managed ISR.

The implementation shall document the following items:

1. For each interrupt, its identifying integer value, whether it can be inhibited or not, and the effects of registering ISRs to non inhibitable interrupts (if this is permitted)

2. Which run-time stack the handle method uses when it executes.

3. Any implementation- or hardware-specific activity that happens before the handle method is invoked (e.g., reading device registers, acknowledging devices).

4. The state (inhibited/uninhibited) of the non-reserved interrupts when the program starts. If some interrupts are uninhibited, a mechanism shall be provided that a program can use to protect itself before it can register the corresponding ISR.

5. The treatment of interrupt occurrences that are generated while the interrupt is inhibited; i.e., whether one or more occurrences are held for later delivery, or all are lost.

6. Whether predefined or implementation-defined exceptions are raised as a result of the occurrence of any interrupt (for example, a hardware trap resulting from a segmentation error), and the mapping between the interrupt and the predefined exceptions.

7. On a multi-processor, the rules governing the delivery of an interrupt occurrence to a particular processor. For example, whether execution of the handle method may spin if the lock of the associated object is held by another processor.

SCJ requires that all code called from *any* method declared within an ISR class that is synchronized on the lock of the ISR object shall not self-suspend. Furthermore, the application should refrain from memory allocations in an outer-nested immortal or mission memory area during the execution of an ISR.

SCJ does not require any further specific restrictions on ISRs. However it requires that the following methods be callable from within an ISR, and therefore these methods shall not self suspend:

- Object.notify and Object.notifyAll,
- all methods of classes that implement the set of raw memory accessor interfaces,
- AperiodicEventHandler.release().

SCJ requires that all methods that can be called from an ISR object shall be annotated with @SCJMaySelfSuspend(false) and shall not be annotated with @SCJMayAllocate({OuterContext}).

SCJ defines the notion of interrupt priorities (see Section 4.7.5). Interrupt priorities shall only be used to define ceiling priorities. All instances of the ManagedInterruptServiceRoutine class should be assigned a ceiling priority that is equal to or higher than the hardware interrupt priority, when it is registered. The normal rules for nested synchronized method calls apply; that is, the ceiling of any object that has a synchronized method that is called from a synchronized method in another object must have a ceiling greater than or equal to the object from which the nested call is made.

The handle method, if not synchronized, shall execute at the hardware interrupt priority.

5.2.2 Level Considerations

.

Level 0

Non-reserved ISRs of any kind are prohibited at Level 0. All interaction with the external embedded environment must be performed in a synchronous manner.

5.2.3 API

Unless indicated otherwise, the classes defined in this section are thread safe

5.2.4 javax.realtime.device.RawByteReader

Declaration

```
@SCJAllowed
 public interface RawByteReader extends javax.realtime.device.RawMemory
```

Since
RTSJ 2.0

Description

A marker for a byte accessor object encapsulating the protocol for reading bytes from raw memory. A byte accessor can always access at least one byte. Each byte is transfered in a single atomic operation. Groups of bytes may be transfered together; however, this is not required.

Objects of this type are created with the method **javax.realtime.device.RawMemoryFactorycreateRawByteReader** and **javax.realtime.device.RawMemoryFactorycreateRawByte** . Each object references a range of elements in the **javax.realtime.device.RawMemoryRegion** starting at the *base address* provided to the factory method. The size provided to the factor method determines the number of elements accessable.

Caching of the memory access is controlled by the factory that created this object. If the memory is not cached, this method guarantees serialized access. In other words, the memory access at the memory occurs in the same order as in the program. Multiple writes to the same location may not be coalesced.

Methods

```
@SCJAllowed
@SCJMayAllocate({})
@SCJMaySelfSuspend(false)
@SCJPhase({
   javax.safetycritical.annotate.Phase.STARTUP,
```

```
    javax.safetycritical.annotate.Phase.INITIALIZATION,
    javax.safetycritical.annotate.Phase.RUN,
    javax.safetycritical.annotate.Phase.CLEANUP })
public int get(int offset, byte [] values)
  throws javax.realtime.OffsetOutOfBoundsException,
      java.lang.NullPointerException
```

Fill values with elements from this instance, where the nth element is at the address: base address + (offset+n) x stride x element size in bytes. Only the bytes in the intersection of the start and end of values and the *base address* and the end of the memory region are transfered. When an exception is thrown, no data is transfered.

offset — of the first byte in the memory region to transfere

values — the array to receive the bytes

returns the number of elements actuall transferred to values

Throws OffsetOutOfBoundsException when offset is negative or greater than or equal to the number of elements in the raw memory region.

Throws NullPointerException when values is null.

```
@SCJAllowed
@SCJMayAllocate({})
@SCJMaySelfSuspend(false)
@SCJPhase({
    javax.safetycritical.annotate.Phase.STARTUP,
    javax.safetycritical.annotate.Phase.INITIALIZATION,
    javax.safetycritical.annotate.Phase.RUN,
    javax.safetycritical.annotate.Phase.CLEANUP })
public int get(int offset, byte [] values, int start, int count)
  throws javax.realtime.OffsetOutOfBoundsException,
      java.lang.ArrayIndexOutOfBoundsException, java.lang.NullPointerException
```

Fill values from index start with elements from this instance, where the nth element is at the address: base address + (offset+n) x stride x element size in bytes. The number of bytes transfered is the minimum of count, the *size* of the memory region minus offset, and length of values minus start. When an exception is thrown, no data is transfered.

offset — of the first byte in the memory region to transfere

values — the array to receive the bytes

start — the first index in array to fill

count — the maximum number of bytes to copy

returns the number of bytes actually transfered.

Throws OffsetOutOfBoundsException when offset is negative or either offset or offset + count is greater than or equal to the size of this raw memory area.

Throws ArrayIndexOutOfBoundsException when start is negative or either start or start + count is greater than or equal to the size of values.

Throws NullPointerException when values is null or count is negative.

```
@SCJAllowed
@SCJMayAllocate({})
@SCJMaySelfSuspend(false)
@SCJPhase({
    javax.safetycritical.annotate.Phase.STARTUP,
    javax.safetycritical.annotate.Phase.INITIALIZATION,
    javax.safetycritical.annotate.Phase.RUN,
    javax.safetycritical.annotate.Phase.CLEANUP })
public byte getByte(int offset)
  throws javax.realtime.OffsetOutOfBoundsException
```

Get the value at the address: base address + offset x stride x element size in bytes. When an exception is thrown, no data is transfered.

offset — of byte in the memory region starting from the address specified in the associated factory method.

returns the value at the address specified.

Throws OffsetOutOfBoundsException when offset is negative or greater than or equal to the number of elements in the raw memory region.

```
@SCJAllowed
@SCJMayAllocate({})
@SCJMaySelfSuspend(false)
@SCJPhase({
    javax.safetycritical.annotate.Phase.STARTUP,
    javax.safetycritical.annotate.Phase.INITIALIZATION,
    javax.safetycritical.annotate.Phase.RUN,
    javax.safetycritical.annotate.Phase.CLEANUP })
public byte getByte( )
```

Get the value at the first position referenced by this instance, i.e., the value at its start address. This operation must be atomic with respect to all other raw memory accesses to the address.

returns the value at the *base address*.

5.2.5 javax.realtime.device.RawByteWriter

Declaration

```
@SCJAllowed
 public interface RawByteWriter extends javax.realtime.device.RawMemory
```

Since
RTSJ 2.0

Description

A marker for a byte accessor object encapsulating the protocol for writing bytes to raw memory. A byte accessor can always access at least one byte. Each byte is transfered in a single atomic operation. Groups of bytes may be transfered together; however, this is not required.

Objects of this type are created with the method **javax.realtime.device.RawMemoryFactorycreateRawByteWriter** and **javax.realtime.device.RawMemoryFactorycreateRawByte** . Each object references a range of elements in the **javax.realtime.device.RawMemoryRegion** starting at the *base address* provided to the factory method. The size provided to the factor method determines the number of elements accessable.

Caching of the memory access is controlled by the factory that created this object. If the memory is not cached, this method guarantees serialized access. In other words, the memory access at the memory occurs in the same order as in the program. Multiple writes to the same location may not be coalesced.

Methods

```
@SCJAllowed
@SCJMayAllocate({})
@SCJMaySelfSuspend(false)
@SCJPhase({
   javax.safetycritical.annotate.Phase.STARTUP,
   javax.safetycritical.annotate.Phase.INITIALIZATION,
   javax.safetycritical.annotate.Phase.RUN,
   javax.safetycritical.annotate.Phase.CLEANUP })
public int set(int offset, byte [] values)
  throws javax.realtime.OffsetOutOfBoundsException,
     java.lang.NullPointerException
```

Copy from values to the memory region from index start,to elements where the nth element is at the address: base address + (offset+n) x stride x element size in bytes. Only the bytes in the intersection of values and the end of the memory region are transfered. When an exception is thrown, no data is transfered.

offset — of first byte in the memory region to be set.

values — is the source of the data to write.

returns the number of elements actually transferred to values

Throws OffsetOutOfBoundsException when offset is negative or greater than or equal to the number of elements in the raw memory region.

Throws NullPointerException when values is null.

```
@SCJAllowed
@SCJMayAllocate({})
@SCJMaySelfSuspend(false)
@SCJPhase({
    javax.safetycritical.annotate.Phase.STARTUP,
    javax.safetycritical.annotate.Phase.INITIALIZATION,
    javax.safetycritical.annotate.Phase.RUN,
    javax.safetycritical.annotate.Phase.CLEANUP })
public int set(int offset, byte [] values, int start, int count)
  throws javax.realtime.OffsetOutOfBoundsException,
      java.lang.ArrayIndexOutOfBoundsException, java.lang.NullPointerException
```

Copy values to the memory region, where offset is first byte in the memory region to write and start is the first index in values from which to read. The number of bytes transfered is the minimum of count, the *size* of the memory region minus offset, and length of values minus start. When an exception is thrown, no data is transfered.

offset — of the first byte in the memory region to set

values — the array from which to retrieve the bytes

start — the first index in array to copy

count — the maximum number of bytes to copy

returns the number of bytes actually transfered.

Throws OffsetOutOfBoundsException when offset is negative or either offset or offset + count is greater than or equal to the size of this raw memory area.

Throws ArrayIndexOutOfBoundsException when start is negative or either start or start + count is greater than or equal to the size of values.

Throws NullPointerException when values is null.

```
@SCJAllowed
@SCJMayAllocate({})
@SCJMaySelfSuspend(false)
@SCJPhase({
```

```
    javax.safetycritical.annotate.Phase.STARTUP,
    javax.safetycritical.annotate.Phase.INITIALIZATION,
    javax.safetycritical.annotate.Phase.RUN,
    javax.safetycritical.annotate.Phase.CLEANUP })
public void setByte(int offset, byte value)
  throws javax.realtime.OffsetOutOfBoundsException
```

Set the value of the n^{th} element referenced by this instance, where n is offset and the address is *base address* + offset x *size of Byte*. This operation must be atomic with respect to all other raw memory accesses to the address. When an exception is thrown, no data is transfered.

offset — of byte in the memory region.

value — is the new value for the element.

Throws OffsetOutOfBoundsException when offset is negative or greater than or equal to the number of elements in the raw memory region.

```
@SCJAllowed
@SCJMayAllocate({})
@SCJMaySelfSuspend(false)
@SCJPhase({
    javax.safetycritical.annotate.Phase.STARTUP,
    javax.safetycritical.annotate.Phase.INITIALIZATION,
    javax.safetycritical.annotate.Phase.RUN,
    javax.safetycritical.annotate.Phase.CLEANUP })
public void setByte(byte value)
```

Set the value at the first position referenced by this instance, i.e., the value at its start address. This operation must be atomic with respect to all other raw memory accesses to the address.

value — is the new value for the element.

5.2.6 javax.realtime.device.RawByte

Declaration

```
@SCJAllowed
 public interface RawByte extends javax.realtime.device.RawByteReader,
     javax.realtime.device.RawByteWriter
```

Since
RTSJ 2.0

Description

A marker for an object that can be used to access to a single byte. Read and write access to that byte is checked by the factory that creates the instance; therefore, no access checking is provided by this interface, only bounds checking.

5.2.7 javax.realtime.device.RawShortReader

Declaration

```
@SCJAllowed
 public interface RawShortReader extends javax.realtime.device.RawMemory
```

Since
RTSJ 2.0

Description

A marker for a short accessor object encapsulating the protocol for reading shorts from raw memory. A short accessor can always access at least one short. Each short is transfered in a single atomic operation. Groups of shorts may be transfered together; however, this is not required.

Objects of this type are created with the method **javax.realtime.device.RawMemoryFactorycreateRawShortReader** and **javax.realtime.device.RawMemoryFactorycreateRawShort** . Each object references a range of elements in the **javax.realtime.device.RawMemoryRegion** starting at the *base address* provided to the factory method. The size provided to the factor method determines the number of elements accessable.

Caching of the memory access is controlled by the factory that created this object. If the memory is not cached, this method guarantees serialized access. In other words, the memory access at the memory occurs in the same order as in the program. Multiple writes to the same location may not be coalesced.

Methods

```
@SCJAllowed
@SCJMayAllocate({})
@SCJMaySelfSuspend(false)
@SCJPhase({
    javax.safetycritical.annotate.Phase.STARTUP,
    javax.safetycritical.annotate.Phase.INITIALIZATION,
    javax.safetycritical.annotate.Phase.RUN,
    javax.safetycritical.annotate.Phase.CLEANUP })
public int get(int offset, short [] values)
  throws javax.realtime.OffsetOutOfBoundsException,
      java.lang.NullPointerException
```

Fill values with elements from this instance, where the nth element is at the address: base address + (offset+n) x stride x element size in bytes. Only the shorts in the intersection of the start and end of values and the *base address* and the end of the memory region are transfered. When an exception is thrown, no data is transfered.

offset — of the first short in the memory region to transfere

values — the array to receive the shorts

returns the number of elements actuall transferred to values

Throws OffsetOutOfBoundsException when offset is negative or greater than or equal to the number of elements in the raw memory region.

Throws NullPointerException when values is null.

```
@SCJAllowed
@SCJMayAllocate({})
@SCJMaySelfSuspend(false)
@SCJPhase({
    javax.safetycritical.annotate.Phase.STARTUP,
    javax.safetycritical.annotate.Phase.INITIALIZATION,
    javax.safetycritical.annotate.Phase.RUN,
    javax.safetycritical.annotate.Phase.CLEANUP })
public int get(int offset, short [] values, int start, int count)
  throws javax.realtime.OffsetOutOfBoundsException,
      java.lang.ArrayIndexOutOfBoundsException, java.lang.NullPointerException
```

Fill values from index start with elements from this instance, where the nth element is at the address: base address + (offset+n) x stride x element size in bytes. The number of bytes transfered is the minimum of count, the *size* of the memory region minus offset, and length of values minus start. When an exception is thrown, no data is transfered.

offset — of the first short in the memory region to transfere

values — the array to receive the shorts

start — the first index in array to fill

count — the maximum number of shorts to copy

returns the number of shorts actually transfered.

Throws OffsetOutOfBoundsException when offset is negative or either offset or offset + count is greater than or equal to the size of this raw memory area.

Throws ArrayIndexOutOfBoundsException when start is negative or either start or start + count is greater than or equal to the size of values.

Throws NullPointerException when values is null or count is negative.

```
@SCJAllowed
@SCJMayAllocate({})
@SCJMaySelfSuspend(false)
@SCJPhase({
    javax.safetycritical.annotate.Phase.STARTUP,
    javax.safetycritical.annotate.Phase.INITIALIZATION,
    javax.safetycritical.annotate.Phase.RUN,
    javax.safetycritical.annotate.Phase.CLEANUP })
public short getShort(int offset)
  throws javax.realtime.OffsetOutOfBoundsException
```

Get the value at the address: base address + offset x stride x element size in bytes. When an exception is thrown, no data is transfered.

offset — of short in the memory region starting from the address specified in the associated factory method.

returns the value at the address specified.

Throws OffsetOutOfBoundsException when offset is negative or greater than or equal to the number of elements in the raw memory region.

```
@SCJAllowed
@SCJMayAllocate({})
@SCJMaySelfSuspend(false)
@SCJPhase({
    javax.safetycritical.annotate.Phase.STARTUP,
    javax.safetycritical.annotate.Phase.INITIALIZATION,
    javax.safetycritical.annotate.Phase.RUN,
    javax.safetycritical.annotate.Phase.CLEANUP })
public short getShort( )
```

Get the value at the first position referenced by this instance, i.e., the value at its start address. This operation must be atomic with respect to all other raw memory accesses to the address.

returns the value at the *base address*.

5.2.8 javax.realtime.device.RawShortWriter

Declaration

```
@SCJAllowed
 public interface RawShortWriter extends javax.realtime.device.RawMemory
```

Since
RTSJ 2.0

Description

A marker for a short accessor object encapsulating the protocol for writing shorts to raw memory. A short accessor can always access at least one short. Each short is transfered in a single atomic operation. Groups of shorts may be transfered together; however, this is not required.

Objects of this type are created with the method **javax.realtime.device.RawMemoryFactorycreateRawShortWriter** and **javax.realtime.device.RawMemoryFactorycreateRawShort** . Each object references a range of elements in the **javax.realtime.device.RawMemoryRegion** starting at the *base address* provided to the factory method. The size provided to the factor method determines the number of elements accessable.

Caching of the memory access is controlled by the factory that created this object. If the memory is not cached, this method guarantees serialized access. In other words, the memory access at the memory occurs in the same order as in the program. Multiple writes to the same location may not be coalesced.

Methods

```
@SCJAllowed
@SCJMayAllocate({})
@SCJMaySelfSuspend(false)
@SCJPhase({
   javax.safetycritical.annotate.Phase.STARTUP,
   javax.safetycritical.annotate.Phase.INITIALIZATION,
   javax.safetycritical.annotate.Phase.RUN,
   javax.safetycritical.annotate.Phase.CLEANUP })
public int set(int offset, short [] values)
  throws javax.realtime.OffsetOutOfBoundsException,
     java.lang.NullPointerException
```

Copy from values to the memory region from index start,to elements where the nth element is at the address: base address + (offset+n) x stride x element size in bytes. Only the shorts in the intersection of values and the end of the memory region are transfered. When an exception is thrown, no data is transfered.

offset — of first short in the memory region to be set.

values — is the source of the data to write.

returns the number of elements actually transferred to values

Throws OffsetOutOfBoundsException when offset is negative or greater than or equal to the number of elements in the raw memory region.

Throws NullPointerException when values is null.

```
@SCJAllowed
@SCJMayAllocate({})
@SCJMaySelfSuspend(false)
@SCJPhase({
    javax.safetycritical.annotate.Phase.STARTUP,
    javax.safetycritical.annotate.Phase.INITIALIZATION,
    javax.safetycritical.annotate.Phase.RUN,
    javax.safetycritical.annotate.Phase.CLEANUP })
public int set(int offset, short [] values, int start, int count)
  throws javax.realtime.OffsetOutOfBoundsException,
     java.lang.ArrayIndexOutOfBoundsException, java.lang.NullPointerException
```

Copy values to the memory region, where offset is first short in the memory region to write and start is the first index in values from which to read. The number of bytes transfered is the minimum of count, the *size* of the memory region minus offset, and length of values minus start. When an exception is thrown, no data is transfered.

offset — of the first short in the memory region to set

values — the array from which to retrieve the shorts

start — the first index in array to copy

count — the maximum number of shorts to copy

returns the number of shorts actually transfered.

Throws OffsetOutOfBoundsException when offset is negative or either offset or offset + count is greater than or equal to the size of this raw memory area.

Throws ArrayIndexOutOfBoundsException when start is negative or either start or start + count is greater than or equal to the size of values.

Throws NullPointerException when values is null.

```
@SCJAllowed
@SCJMayAllocate({})
@SCJMaySelfSuspend(false)
@SCJPhase({
    javax.safetycritical.annotate.Phase.STARTUP,
    javax.safetycritical.annotate.Phase.INITIALIZATION,
    javax.safetycritical.annotate.Phase.RUN,
    javax.safetycritical.annotate.Phase.CLEANUP })
public void setShort(int offset, short value)
  throws javax.realtime.OffsetOutOfBoundsException
```

Set the value of the n^{th} element referenced by this instance, where n is offset and the address is *base address* + offset x *size of Short*. This operation must be atomic with respect to all other raw memory accesses to the address. When an exception is thrown, no data is transfered.

offset — of short in the memory region.

value — is the new value for the element.

Throws OffsetOutOfBoundsException when offset is negative or greater than or equal to the number of elements in the raw memory region.

```
@SCJAllowed
@SCJMayAllocate({})
@SCJMaySelfSuspend(false)
@SCJPhase({
    javax.safetycritical.annotate.Phase.STARTUP,
    javax.safetycritical.annotate.Phase.INITIALIZATION,
    javax.safetycritical.annotate.Phase.RUN,
    javax.safetycritical.annotate.Phase.CLEANUP })
public void setShort(short value)
```

Set the value at the first position referenced by this instance, i.e., the value at its start address. This operation must be atomic with respect to all other raw memory accesses to the address.

value — is the new value for the element.

5.2.9 javax.realtime.device.RawShort

Declaration

```
@SCJAllowed
 public interface RawShort extends javax.realtime.device.RawShortReader,
     javax.realtime.device.RawShortWriter
```

Since
RTSJ 2.0

Description

A marker for an object that can be used to access to a single short. Read and write access to that short is checked by the factory that creates the instance; therefore, no access checking is provided by this interface, only bounds checking.

5.2.10 javax.realtime.device.RawIntReader

Declaration

```
@SCJAllowed
 public interface RawIntReader extends javax.realtime.device.RawMemory
```

Since
RTSJ 2.0

Description

A marker for a int accessor object encapsulating the protocol for reading ints from raw memory. A int accessor can always access at least one int. Each int is transfered in a single atomic operation. Groups of ints may be transfered together; however, this is not required.

Objects of this type are created with the method **javax.realtime.device.RawMemoryFactorycreateRawIntReader** and **javax.realtime.device.RawMemoryFactorycreateRawInt** . Each object references a range of elements in the **javax.realtime.device.RawMemoryRegion** starting at the *base address* provided to the factory method. The size provided to the factor method determines the number of elements accessable.

Caching of the memory access is controlled by the factory that created this object. If the memory is not cached, this method guarantees serialized access. In other words, the memory access at the memory occurs in the same order as in the program. Multiple writes to the same location may not be coalesced.

Methods

```
@SCJAllowed
@SCJMayAllocate({})
@SCJMaySelfSuspend(false)
@SCJPhase({
    javax.safetycritical.annotate.Phase.STARTUP,
    javax.safetycritical.annotate.Phase.INITIALIZATION,
    javax.safetycritical.annotate.Phase.RUN,
    javax.safetycritical.annotate.Phase.CLEANUP })
public int get(int offset, int [] values)
  throws javax.realtime.OffsetOutOfBoundsException,
      java.lang.NullPointerException
```

Fill values with elements from this instance, where the nth element is at the address: base address + (offset+n) x stride x element size in bytes. Only the ints in the intersection of the start and end of values and the *base address* and the end of the memory region are transfered. When an exception is thrown, no data is transfered.

offset — of the first int in the memory region to transfere

values — the array to receive the ints

returns the number of elements actuall transferred to values

Throws OffsetOutOfBoundsException when offset is negative or greater than or equal to the number of elements in the raw memory region.

Throws NullPointerException when values is null.

```
@SCJAllowed
@SCJMayAllocate({})
@SCJMaySelfSuspend(false)
@SCJPhase({
    javax.safetycritical.annotate.Phase.STARTUP,
    javax.safetycritical.annotate.Phase.INITIALIZATION,
    javax.safetycritical.annotate.Phase.RUN,
    javax.safetycritical.annotate.Phase.CLEANUP })
public int get(int offset, int [] values, int start, int count)
  throws javax.realtime.OffsetOutOfBoundsException,
      java.lang.ArrayIndexOutOfBoundsException, java.lang.NullPointerException
```

Fill values from index start with elements from this instance, where the nth element is at the address: base address + (offset+n) x stride x element size in bytes. The number of bytes transfered is the minimum of count, the *size* of the memory region minus offset, and length of values minus start. When an exception is thrown, no data is transfered.

offset — of the first int in the memory region to transfere

values — the array to receive the ints

start — the first index in array to fill

count — the maximum number of ints to copy

returns the number of ints actually transfered.

Throws OffsetOutOfBoundsException when offset is negative or either offset or offset + count is greater than or equal to the size of this raw memory area.

Throws ArrayIndexOutOfBoundsException when start is negative or either start or start + count is greater than or equal to the size of values.

Throws NullPointerException when values is null or count is negative.

```
@SCJAllowed
@SCJMayAllocate({})
@SCJMaySelfSuspend(false)
@SCJPhase({
```

```
    javax.safetycritical.annotate.Phase.STARTUP,
    javax.safetycritical.annotate.Phase.INITIALIZATION,
    javax.safetycritical.annotate.Phase.RUN,
    javax.safetycritical.annotate.Phase.CLEANUP })
public int getInt(int offset)
  throws javax.realtime.OffsetOutOfBoundsException
```

Get the value at the address: base address + offset x stride x element size in bytes. When an exception is thrown, no data is transfered.

offset — of int in the memory region starting from the address specified in the associated factory method.

returns the value at the address specified.

Throws OffsetOutOfBoundsException when offset is negative or greater than or equal to the number of elements in the raw memory region.

```
@SCJAllowed
@SCJMayAllocate({})
@SCJMaySelfSuspend(false)
@SCJPhase({
    javax.safetycritical.annotate.Phase.STARTUP,
    javax.safetycritical.annotate.Phase.INITIALIZATION,
    javax.safetycritical.annotate.Phase.RUN,
    javax.safetycritical.annotate.Phase.CLEANUP })
public int getInt( )
```

Get the value at the first position referenced by this instance, i.e., the value at its start address. This operation must be atomic with respect to all other raw memory accesses to the address.

returns the value at the *base address*.

5.2.11 javax.realtime.device.RawIntWriter

Declaration

```
@SCJAllowed
 public interface RawIntWriter extends javax.realtime.device.RawMemory
```

Since
RTSJ 2.0

Description

A marker for a int accessor object encapsulating the protocol for writing ints to raw memory. A int accessor can always access at least one int. Each int is transfered in a single atomic operation. Groups of ints may be transfered together; however, this is not required.

Objects of this type are created with the method **javax.realtime.device.RawMemoryFactorycreateRawIntWriter** and **javax.realtime.device.RawMemoryFactorycreateRawInt** . Each object references a range of elements in the **javax.realtime.device.RawMemoryRegion** starting at the *base address* provided to the factory method. The size provided to the factor method determines the number of elements accessable.

Caching of the memory access is controlled by the factory that created this object. If the memory is not cached, this method guarantees serialized access. In other words, the memory access at the memory occurs in the same order as in the program. Multiple writes to the same location may not be coalesced.

Methods

```
@SCJAllowed
@SCJMayAllocate({})
@SCJMaySelfSuspend(false)
@SCJPhase({
    javax.safetycritical.annotate.Phase.STARTUP,
    javax.safetycritical.annotate.Phase.INITIALIZATION,
    javax.safetycritical.annotate.Phase.RUN,
    javax.safetycritical.annotate.Phase.CLEANUP })
public int set(int offset, int [] values)
  throws javax.realtime.OffsetOutOfBoundsException,
      java.lang.NullPointerException
```

Copy from values to the memory region from index start,to elements where the nth element is at the address: base address + (offset+n) x stride x element size in bytes. Only the ints in the intersection of values and the end of the memory region are transfered. When an exception is thrown, no data is transfered.

offset — of first int in the memory region to be set.

values — is the source of the data to write.

returns the number of elements actually transferred to values

Throws OffsetOutOfBoundsException when offset is negative or greater than or equal to the number of elements in the raw memory region.

Throws NullPointerException when values is null.

```
@SCJAllowed
@SCJMayAllocate({})
@SCJMaySelfSuspend(false)
@SCJPhase({
    javax.safetycritical.annotate.Phase.STARTUP,
    javax.safetycritical.annotate.Phase.INITIALIZATION,
    javax.safetycritical.annotate.Phase.RUN,
    javax.safetycritical.annotate.Phase.CLEANUP })
public int set(int offset, int [] values, int start, int count)
  throws javax.realtime.OffsetOutOfBoundsException,
     java.lang.ArrayIndexOutOfBoundsException, java.lang.NullPointerException
```

Copy values to the memory region, where offset is first int in the memory region to write and start is the first index in values from which to read. The number of bytes transfered is the minimum of count, the *size* of the memory region minus offset, and length of values minus start. When an exception is thrown, no data is transfered.

offset — of the first int in the memory region to set

values — the array from which to retrieve the ints

start — the first index in array to copy

count — the maximum number of ints to copy

returns the number of ints actually transfered.

Throws OffsetOutOfBoundsException when offset is negative or either offset or offset + count is greater than or equal to the size of this raw memory area.

Throws ArrayIndexOutOfBoundsException when start is negative or either start or start + count is greater than or equal to the size of values.

Throws NullPointerException when values is null.

```
@SCJAllowed
@SCJMayAllocate({})
@SCJMaySelfSuspend(false)
@SCJPhase({
    javax.safetycritical.annotate.Phase.STARTUP,
    javax.safetycritical.annotate.Phase.INITIALIZATION,
    javax.safetycritical.annotate.Phase.RUN,
    javax.safetycritical.annotate.Phase.CLEANUP })
public void setInt(int offset, int value)
  throws javax.realtime.OffsetOutOfBoundsException
```

Set the value of the n^{th} element referenced by this instance, where n is offset and the address is *base address* + offset x *size of Int*. This operation must be atomic with respect to all other raw memory accesses to the address. When an exception is thrown, no data is transfered.

offset — of int in the memory region.

value — is the new value for the element.

Throws OffsetOutOfBoundsException when offset is negative or greater than or equal to the number of elements in the raw memory region.

```
@SCJAllowed
@SCJMayAllocate({})
@SCJMaySelfSuspend(false)
@SCJPhase({
    javax.safetycritical.annotate.Phase.STARTUP,
    javax.safetycritical.annotate.Phase.INITIALIZATION,
    javax.safetycritical.annotate.Phase.RUN,
    javax.safetycritical.annotate.Phase.CLEANUP })
public void setInt(int value)
```

Set the value at the first position referenced by this instance, i.e., the value at its start address. This operation must be atomic with respect to all other raw memory accesses to the address.

value — is the new value for the element.

5.2.12 javax.realtime.device.RawInt

Declaration

```
@SCJAllowed
 public interface RawInt extends javax.realtime.device.RawIntReader,
     javax.realtime.device.RawIntWriter
```

Since
RTSJ 2.0

Description

A marker for an object that can be used to access to a single int. Read and write access to that int is checked by the factory that creates the instance; therefore, no access checking is provided by this interface, only bounds checking.

5.2.13 javax.realtime.device.RawLongReader

Declaration

```
@SCJAllowed
 public interface RawLongReader extends javax.realtime.device.RawMemory
```

Since
RTSJ 2.0

Description

A marker for a long accessor object encapsulating the protocol for reading longs from raw memory. A long accessor can always access at least one long. Each long is transfered in a single atomic operation. Groups of longs may be transfered together; however, this is not required.

Objects of this type are created with the method **javax.realtime.device.RawMemoryFactorycreateRawLongReader** and **javax.realtime.device.RawMemoryFactorycreateRawLong** . Each object references a range of elements in the **javax.realtime.device.RawMemoryRegion** starting at the *base address* provided to the factory method. The size provided to the factor method determines the number of elements accessable.

Caching of the memory access is controlled by the factory that created this object. If the memory is not cached, this method guarantees serialized access. In other words, the memory access at the memory occurs in the same order as in the program. Multiple writes to the same location may not be coalesced.

Methods

```
@SCJAllowed
@SCJMayAllocate({})
@SCJMaySelfSuspend(false)
@SCJPhase({
    javax.safetycritical.annotate.Phase.STARTUP,
    javax.safetycritical.annotate.Phase.INITIALIZATION,
    javax.safetycritical.annotate.Phase.RUN,
    javax.safetycritical.annotate.Phase.CLEANUP })
public int get(int offset, long [] values)
  throws javax.realtime.OffsetOutOfBoundsException,
      java.lang.NullPointerException
```

Fill values with elements from this instance, where the nth element is at the address: base address + (offset+n) x stride x element size in bytes. Only the longs in the intersection of the start and end of values and the *base address* and the end of the memory region are transfered. When an exception is thrown, no data is transfered.

offset — of the first long in the memory region to transfere

values — the array to receive the longs

returns the number of elements actuall transferred to values

Throws OffsetOutOfBoundsException when offset is negative or greater than or equal to the number of elements in the raw memory region.

Throws NullPointerException when values is null.

```
@SCJAllowed
@SCJMayAllocate({})
@SCJMaySelfSuspend(false)
@SCJPhase({
    javax.safetycritical.annotate.Phase.STARTUP,
    javax.safetycritical.annotate.Phase.INITIALIZATION,
    javax.safetycritical.annotate.Phase.RUN,
    javax.safetycritical.annotate.Phase.CLEANUP })
public int get(int offset, long [] values, int start, int count)
  throws javax.realtime.OffsetOutOfBoundsException,
      java.lang.ArrayIndexOutOfBoundsException, java.lang.NullPointerException
```

Fill values from index start with elements from this instance, where the nth element is at the address: base address + (offset+n) x stride x element size in bytes. The number of bytes transfered is the minimum of count, the *size* of the memory region minus offset, and length of values minus start. When an exception is thrown, no data is transfered.

offset — of the first long in the memory region to transfere

values — the array to receive the longs

start — the first index in array to fill

count — the maximum number of longs to copy

returns the number of longs actually transfered.

Throws OffsetOutOfBoundsException when offset is negative or either offset or offset + count is greater than or equal to the size of this raw memory area.

Throws ArrayIndexOutOfBoundsException when start is negative or either start or start + count is greater than or equal to the size of values.

Throws NullPointerException when values is null or count is negative.

```
@SCJAllowed
@SCJMayAllocate({})
@SCJMaySelfSuspend(false)
@SCJPhase({
    javax.safetycritical.annotate.Phase.STARTUP,
    javax.safetycritical.annotate.Phase.INITIALIZATION,
    javax.safetycritical.annotate.Phase.RUN,
    javax.safetycritical.annotate.Phase.CLEANUP })
public long getLong(int offset)
  throws javax.realtime.OffsetOutOfBoundsException
```

Get the value at the address: base address + offset x stride x element size in bytes. When an exception is thrown, no data is transfered.

offset — of long in the memory region starting from the address specified in the associated factory method.

returns the value at the address specified.

Throws OffsetOutOfBoundsException when offset is negative or greater than or equal to the number of elements in the raw memory region.

```
@SCJAllowed
@SCJMayAllocate({})
@SCJMaySelfSuspend(false)
@SCJPhase({
    javax.safetycritical.annotate.Phase.STARTUP,
    javax.safetycritical.annotate.Phase.INITIALIZATION,
    javax.safetycritical.annotate.Phase.RUN,
    javax.safetycritical.annotate.Phase.CLEANUP })
public long getLong( )
```

Get the value at the first position referenced by this instance, i.e., the value at its start address. This operation must be atomic with respect to all other raw memory accesses to the address.

returns the value at the *base address*.

5.2.14 javax.realtime.device.RawLongWriter

Declaration

```
@SCJAllowed
 public interface RawLongWriter extends javax.realtime.device.RawMemory
```

Since
RTSJ 2.0

Description

A marker for a long accessor object encapsulating the protocol for writing longs to raw memory. A long accessor can always access at least one long. Each long is transfered in a single atomic operation. Groups of longs may be transfered together; however, this is not required.

Objects of this type are created with the method **javax.realtime.device.RawMemoryFactorycreateRawLongWriter** and **javax.realtime.device.RawMemoryFactorycreateRawLong** . Each object references a range of elements in the

javax.realtime.device.RawMemoryRegion starting at the *base address* provided to the factory method. The size provided to the factor method determines the number of elements accessable.

Caching of the memory access is controlled by the factory that created this object. If the memory is not cached, this method guarantees serialized access. In other words, the memory access at the memory occurs in the same order as in the program. Multiple writes to the same location may not be coalesced.

Methods

```
@SCJAllowed
@SCJMayAllocate({})
@SCJMaySelfSuspend(false)
@SCJPhase({
    javax.safetycritical.annotate.Phase.STARTUP,
    javax.safetycritical.annotate.Phase.INITIALIZATION,
    javax.safetycritical.annotate.Phase.RUN,
    javax.safetycritical.annotate.Phase.CLEANUP })
public int set(int offset, long [] values)
  throws javax.realtime.OffsetOutOfBoundsException,
      java.lang.NullPointerException
```

Copy from values to the memory region from index start,to elements where the nth element is at the address: base address + (offset+n) x stride x element size in bytes. Only the longs in the intersection of values and the end of the memory region are transfered. When an exception is thrown, no data is transfered.

offset — of first long in the memory region to be set.

values — is the source of the data to write.

returns the number of elements actually transferred to values

Throws OffsetOutOfBoundsException when offset is negative or greater than or equal to the number of elements in the raw memory region.

Throws NullPointerException when values is null.

```
@SCJAllowed
@SCJMayAllocate({})
@SCJMaySelfSuspend(false)
@SCJPhase({
    javax.safetycritical.annotate.Phase.STARTUP,
    javax.safetycritical.annotate.Phase.INITIALIZATION,
    javax.safetycritical.annotate.Phase.RUN,
    javax.safetycritical.annotate.Phase.CLEANUP })
public int set(int offset, long [] values, int start, int count)
  throws javax.realtime.OffsetOutOfBoundsException,
      java.lang.ArrayIndexOutOfBoundsException, java.lang.NullPointerException
```

Copy values to the memory region, where offset is first long in the memory region to write and start is the first index in values from which to read. The number of bytes transfered is the minimum of count, the *size* of the memory region minus offset, and length of values minus start. When an exception is thrown, no data is transfered.

offset — of the first long in the memory region to set

values — the array from which to retrieve the longs

start — the first index in array to copy

count — the maximum number of longs to copy

returns the number of longs actually transfered.

Throws OffsetOutOfBoundsException when offset is negative or either offset or offset + count is greater than or equal to the size of this raw memory area.

Throws ArrayIndexOutOfBoundsException when start is negative or either start or start + count is greater than or equal to the size of values.

Throws NullPointerException when values is null.

```
@SCJAllowed
@SCJMayAllocate({})
@SCJMaySelfSuspend(false)
@SCJPhase({
    javax.safetycritical.annotate.Phase.STARTUP,
    javax.safetycritical.annotate.Phase.INITIALIZATION,
    javax.safetycritical.annotate.Phase.RUN,
    javax.safetycritical.annotate.Phase.CLEANUP })
public void setLong(int offset, long value)
  throws javax.realtime.OffsetOutOfBoundsException
```

Set the value of the n^{th} element referenced by this instance, where n is offset and the address is *base address* + offset x *size of Long*. This operation must be atomic with respect to all other raw memory accesses to the address. When an exception is thrown, no data is transfered.

offset — of long in the memory region.

value — is the new value for the element.

Throws OffsetOutOfBoundsException when offset is negative or greater than or equal to the number of elements in the raw memory region.

```
@SCJAllowed
@SCJMayAllocate({})
@SCJMaySelfSuspend(false)
```

```
@SCJPhase({
   javax.safetycritical.annotate.Phase.STARTUP,
   javax.safetycritical.annotate.Phase.INITIALIZATION,
   javax.safetycritical.annotate.Phase.RUN,
   javax.safetycritical.annotate.Phase.CLEANUP })
public void setLong(long value)
```

Set the value at the first position referenced by this instance, i.e., the value at its start address. This operation must be atomic with respect to all other raw memory accesses to the address.

value — is the new value for the element.

5.2.15 javax.realtime.device.RawLong

Declaration

```
@SCJAllowed
 public interface RawLong extends javax.realtime.device.RawLongReader,
     javax.realtime.device.RawLongWriter
```

Since
RTSJ 2.0

Description

A marker for an object that can be used to access to a single long. Read and write access to that long is checked by the factory that creates the instance; therefore, no access checking is provided by this interface, only bounds checking.

5.2.16 javax.realtime.device.RawMemoryRegion

Declaration

```
@SCJAllowed
 public class RawMemoryRegion extends java.lang.Object
```

Description

RawMemoryRegion is a class for typing raw memory regions. It is returned by the RawMemoryRegionFactory.getRegion methods of the raw memory region factory classes, and it is used with methods such as RawMemoryFactory.createRawByte(RawMemoryRegion, long, int, int) and RawMemoryFactory.createRawDouble(RawMemoryRegion, long, int, int} methods to identify the region from which the application wants to get an accessor instance.

Constructors

```
@SCJAllowed
@SCJMayAllocate({})
@SCJMaySelfSuspend(false)
@SCJPhase({
    javax.safetycritical.annotate.Phase.STARTUP,
    javax.safetycritical.annotate.Phase.INITIALIZATION,
    javax.safetycritical.annotate.Phase.RUN,
    javax.safetycritical.annotate.Phase.CLEANUP})
public RawMemoryRegion(String name)
```

Methods

```
@SCJAllowed
@SCJMayAllocate({})
@SCJMaySelfSuspend(false)
@SCJPhase({
    javax.safetycritical.annotate.Phase.STARTUP,
    javax.safetycritical.annotate.Phase.INITIALIZATION,
    javax.safetycritical.annotate.Phase.RUN,
    javax.safetycritical.annotate.Phase.CLEANUP})
public final java.lang.String getName( )
```

Obtains the name of this region type.

returns the region types name

```
@SCJAllowed
@SCJMaySelfSuspend(false)
@SCJPhase({
    javax.safetycritical.annotate.Phase.STARTUP,
    javax.safetycritical.annotate.Phase.INITIALIZATION,
    javax.safetycritical.annotate.Phase.RUN,
    javax.safetycritical.annotate.Phase.CLEANUP})
@SCJMayAllocate({
    javax.safetycritical.annotate.AllocationContext.CURRENT,
    javax.safetycritical.annotate.AllocationContext.INNER,
    javax.safetycritical.annotate.AllocationContext.OUTER})
public static javax.realtime.device.RawMemoryRegion getRegion(String name)
```

Get a region type when it already exists or creates a new one.

name — of the region

returns the region type object.

```
@SCJAllowed
@SCJMayAllocate({})
@SCJMaySelfSuspend(false)
@SCJPhase({
   javax.safetycritical.annotate.Phase.STARTUP,
   javax.safetycritical.annotate.Phase.INITIALIZATION,
   javax.safetycritical.annotate.Phase.RUN,
   javax.safetycritical.annotate.Phase.CLEANUP})
public static boolean isRawMemoryRegion(String name)
```

Ask whether or not there is a memory region type of a given name.

name — for which to search

returns true when there is one and false otherwise.

```
@SCJAllowed
@SCJMayAllocate({})
@SCJMaySelfSuspend(false)
@SCJPhase({
   javax.safetycritical.annotate.Phase.STARTUP,
   javax.safetycritical.annotate.Phase.INITIALIZATION,
   javax.safetycritical.annotate.Phase.RUN,
   javax.safetycritical.annotate.Phase.CLEANUP})
public final java.lang.String toString( )
```

Gets a printable representation for a Region.

returns the name of this memory region type.

5.2.17 javax.realtime.device.RawMemoryRegionFactory

Declaration

```
@SCJAllowed
 public interface RawMemoryRegionFactory
```

Since
RTSJ 2.0

Description

A class to give an application the ability to provide support for a **javax.realtime.device.RawMemoryRegion** that is not already provided by the standard. An instance of this call can be registered with a **javax.realtime.device.RawMemoryFactory** and provides the object that that factory should return for a given RawMemoryRegion. It is responsible for checking all requests and throwing the proper exception when a request is invalid or the requester is not authorized to make the request.

Methods

```
@SCJAllowed
@SCJMayAllocate({javax.safetycritical.annotate.AllocationContext.CURRENT})
@SCJMaySelfSuspend(false)
@SCJPhase({
    javax.safetycritical.annotate.Phase.STARTUP,
    javax.safetycritical.annotate.Phase.INITIALIZATION,
    javax.safetycritical.annotate.Phase.RUN,
    javax.safetycritical.annotate.Phase.CLEANUP })
public javax.realtime.device.RawByte createRawByte(long base,
  int count,
  int stride)
  throws
      java.lang.SecurityException, javax.realtime.OffsetOutOfBoundsException,
      javax.realtime.SizeOutOfBoundsException,
      javax.realtime.UnsupportedRawMemoryRegionException,
      javax.realtime.MemoryTypeConflictException
```

Create an instance of a class that implements **javax.realtime.device.RawByte** and accesses memory of **javax.realtime.device.RawMemoryRegionFactoryget-Region** in the address range described by base, stride, and count. The actual extent of the memory addressed by the object is stride x *size of RawByte* x count. The object is allocated in the current memory area of the calling thread.

base — The starting physical address accessible through the returned instance.

count — The number of memory elements accessible through the returned instance.

stride — The distance to the next element in mulitple of element count, where a value of 1 means the elements are adjacent in memory.

returns an object that implements **javax.realtime.device.RawByte** and supports access to the specified range in the memory region.

Throws IllegalArgumentException when base is negative, count is not greater than zero, or stride is not greater than zero.

Throws SecurityException when the caller does not have permissions to access the given memory region or the specified range of addresses.

Throws OffsetOutOfBoundsException when base is invalid.

Throws SizeOutOfBoundsException when the memory addressed by the object would extend into an invalid range of memory.

Throws MemoryTypeConflictException when base does not point to memory that matches the type served by this factory.

Since
RTSJ 2.0

```
@SCJAllowed
@SCJMayAllocate({javax.safetycritical.annotate.AllocationContext.CURRENT})
@SCJMaySelfSuspend(false)
@SCJPhase({
    javax.safetycritical.annotate.Phase.STARTUP,
    javax.safetycritical.annotate.Phase.INITIALIZATION,
    javax.safetycritical.annotate.Phase.RUN,
    javax.safetycritical.annotate.Phase.CLEANUP })
public javax.realtime.device.RawByteReader createRawByteReader(long base,
  int count,
  int stride)
  throws
      java.lang.SecurityException, javax.realtime.OffsetOutOfBoundsException,
      javax.realtime.SizeOutOfBoundsException,
      javax.realtime.UnsupportedRawMemoryRegionException,
      javax.realtime.MemoryTypeConflictException
```

Create an instance of a class that implements **javax.realtime.device.RawByteReader** and accesses memory of **javax.realtime.device.RawMemoryRegionFactorygetRegion** in the address range described by base, stride, and count. The actual extent of the memory addressed by the object is stride x *size of RawByteReader* x count. The object is allocated in the current memory area of the calling thread.

base — The starting physical address accessible through the returned instance.

count — The number of memory elements accessible through the returned instance.

stride — The distance to the next element in mulitple of element count, where a value of 1 means the elements are adjacent in memory.

returns an object that implements **javax.realtime.device.RawByteReader** and supports access to the specified range in the memory region.

Throws IllegalArgumentException when base is negative, count is not greater than zero, or stride is not greater than zero.

Throws SecurityException when the caller does not have permissions to access the given memory region or the specified range of addresses.

Throws OffsetOutOfBoundsException when base is invalid.

Throws SizeOutOfBoundsException when the memory addressed by the object would extend into an invalid range of memory.

Throws MemoryTypeConflictException when base does not point to memory that matches the type served by this factory.

Since
RTSJ 2.0

```
@SCJAllowed
@SCJMayAllocate({javax.safetycritical.annotate.AllocationContext.CURRENT})
@SCJMaySelfSuspend(false)
@SCJPhase({
    javax.safetycritical.annotate.Phase.STARTUP,
    javax.safetycritical.annotate.Phase.INITIALIZATION,
    javax.safetycritical.annotate.Phase.RUN,
    javax.safetycritical.annotate.Phase.CLEANUP })
public javax.realtime.device.RawByteWriter createRawByteWriter(long base,
  int count,
  int stride)
  throws
      java.lang.SecurityException, javax.realtime.OffsetOutOfBoundsException,
      javax.realtime.SizeOutOfBoundsException,
      javax.realtime.UnsupportedRawMemoryRegionException,
      javax.realtime.MemoryTypeConflictException
```

Create an instance of a class that implements **javax.realtime.device.RawByteWriter** and accesses memory of **javax.realtime.device.RawMemoryRegionFactorygetRegion** in the address range described by base, stride, and count. The actual extent of the memory addressed by the object is stride x *size of RawByteWriter* x count. The object is allocated in the current memory area of the calling thread.

base — The starting physical address accessible through the returned instance.

count — The number of memory elements accessible through the returned instance.

stride — The distance to the next element in mulitple of element count, where a value of 1 means the elements are adjacent in memory.

returns an object that implements **javax.realtime.device.RawByteWriter** and supports access to the specified range in the memory region.

Throws IllegalArgumentException when base is negative, count is not greater than zero, or stride is not greater than zero.

Throws SecurityException when the caller does not have permissions to access the given memory region or the specified range of addresses.

Throws OffsetOutOfBoundsException when base is invalid.

Throws SizeOutOfBoundsException when the memory addressed by the object would extend into an invalid range of memory.

Throws MemoryTypeConflictException when base does not point to memory that matches the type served by this factory.

Since
RTSJ 2.0

```
@SCJAllowed
@SCJMayAllocate({javax.safetycritical.annotate.AllocationContext.CURRENT})
@SCJMaySelfSuspend(false)
@SCJPhase({
    javax.safetycritical.annotate.Phase.STARTUP,
    javax.safetycritical.annotate.Phase.INITIALIZATION,
    javax.safetycritical.annotate.Phase.RUN,
    javax.safetycritical.annotate.Phase.CLEANUP })
public javax.realtime.device.RawDouble createRawDouble(long base,
  int count,
  int stride)
  throws
      java.lang.SecurityException, javax.realtime.OffsetOutOfBoundsException,
      javax.realtime.SizeOutOfBoundsException,
      javax.realtime.UnsupportedRawMemoryRegionException,
      javax.realtime.MemoryTypeConflictException
```

Create an instance of a class that implements **javax.realtime.device.RawDouble** and accesses memory of **javax.realtime.device.RawMemoryRegionFactoryget-Region** in the address range described by base, stride, and count. The actual extent of the memory addressed by the object is stride x *size of RawDouble* x count. The object is allocated in the current memory area of the calling thread.

base — The starting physical address accessible through the returned instance.

count — The number of memory elements accessible through the returned instance.

stride — The distance to the next element in mulitple of element count, where a value of 1 means the elements are adjacent in memory.

returns an object that implements **javax.realtime.device.RawDouble** and supports access to the specified range in the memory region.

Throws IllegalArgumentException when base is negative, count is not greater than zero, or stride is not greater than zero.

Throws SecurityException when the caller does not have permissions to access the given memory region or the specified range of addresses.

Throws OffsetOutOfBoundsException when base is invalid.

Throws SizeOutOfBoundsException when the memory addressed by the object would extend into an invalid range of memory.

Throws MemoryTypeConflictException when base does not point to memory that matches the type served by this factory.

Since
RTSJ 2.0

```
@SCJAllowed
@SCJMayAllocate({javax.safetycritical.annotate.AllocationContext.CURRENT})
@SCJMaySelfSuspend(false)
@SCJPhase({
    javax.safetycritical.annotate.Phase.STARTUP,
    javax.safetycritical.annotate.Phase.INITIALIZATION,
    javax.safetycritical.annotate.Phase.RUN,
    javax.safetycritical.annotate.Phase.CLEANUP })
public javax.realtime.device.RawDoubleReader createRawDoubleReader(long base,
  int count,
  int stride)
  throws
      java.lang.SecurityException, javax.realtime.OffsetOutOfBoundsException,
      javax.realtime.SizeOutOfBoundsException,
      javax.realtime.UnsupportedRawMemoryRegionException,
      javax.realtime.MemoryTypeConflictException
```

Create an instance of a class that implements **javax.realtime.device.RawDoubleReader** and accesses memory of **javax.realtime.device.RawMemoryRegionFactorygetRegion** in the address range described by base, stride, and count. The actual extent of the memory addressed by the object is stride x *size of RawDoubleReader* x count. The object is allocated in the current memory area of the calling thread.

base — The starting physical address accessible through the returned instance.

count — The number of memory elements accessible through the returned instance.

stride — The distance to the next element in mulitple of element count, where a value of 1 means the elements are adjacent in memory.

returns an object that implements **javax.realtime.device.RawDoubleReader** and supports access to the specified range in the memory region.

Throws IllegalArgumentException when base is negative, count is not greater than zero, or stride is not greater than zero.

Throws SecurityException when the caller does not have permissions to access the given memory region or the specified range of addresses.

Throws OffsetOutOfBoundsException when base is invalid.

Throws SizeOutOfBoundsException when the memory addressed by the object would extend into an invalid range of memory.

Throws MemoryTypeConflictException when base does not point to memory that matches the type served by this factory.

Since
RTSJ 2.0

```
@SCJAllowed
@SCJMayAllocate({javax.safetycritical.annotate.AllocationContext.CURRENT})
@SCJMaySelfSuspend(false)
@SCJPhase({
    javax.safetycritical.annotate.Phase.STARTUP,
    javax.safetycritical.annotate.Phase.INITIALIZATION,
    javax.safetycritical.annotate.Phase.RUN,
    javax.safetycritical.annotate.Phase.CLEANUP })
public javax.realtime.device.RawDoubleWriter createRawDoubleWriter(long base,
  int count,
  int stride)
  throws
      java.lang.SecurityException, javax.realtime.OffsetOutOfBoundsException,
      javax.realtime.SizeOutOfBoundsException,
      javax.realtime.UnsupportedRawMemoryRegionException,
      javax.realtime.MemoryTypeConflictException
```

Create an instance of a class that implements **javax.realtime.device.RawDoubleWriter** and accesses memory of **javax.realtime.device.RawMemoryRegionFactorygetRegion** in the address range described by base, stride, and count. The actual extent of the memory addressed by the object is stride x *size of RawDoubleWriter* x count. The object is allocated in the current memory area of the calling thread.

base — The starting physical address accessible through the returned instance.

count — The number of memory elements accessible through the returned instance.

stride — The distance to the next element in mulitple of element count, where a value of 1 means the elements are adjacent in memory.

returns an object that implements **javax.realtime.device.RawDoubleWriter** and supports access to the specified range in the memory region.

Throws IllegalArgumentException when base is negative, count is not greater than zero, or stride is not greater than zero.

Throws SecurityException when the caller does not have permissions to access the given memory region or the specified range of addresses.

Throws OffsetOutOfBoundsException when base is invalid.

Throws SizeOutOfBoundsException when the memory addressed by the object would extend into an invalid range of memory.

Throws MemoryTypeConflictException when base does not point to memory that matches the type served by this factory.

Since
RTSJ 2.0

```
@SCJAllowed
@SCJMayAllocate({javax.safetycritical.annotate.AllocationContext.CURRENT})
@SCJMaySelfSuspend(false)
@SCJPhase({
    javax.safetycritical.annotate.Phase.STARTUP,
    javax.safetycritical.annotate.Phase.INITIALIZATION,
    javax.safetycritical.annotate.Phase.RUN,
    javax.safetycritical.annotate.Phase.CLEANUP })
public javax.realtime.device.RawFloat createRawFloat(long base,
  int count,
  int stride)
  throws
      java.lang.SecurityException, javax.realtime.OffsetOutOfBoundsException,
      javax.realtime.SizeOutOfBoundsException,
      javax.realtime.UnsupportedRawMemoryRegionException,
      javax.realtime.MemoryTypeConflictException
```

Create an instance of a class that implements **javax.realtime.device.RawFloat** and accesses memory of **javax.realtime.device.RawMemoryRegionFactoryget-Region** in the address range described by base, stride, and count. The actual extent of the memory addressed by the object is stride x *size of RawFloat* x count. The object is allocated in the current memory area of the calling thread.

base — The starting physical address accessible through the returned instance.

count — The number of memory elements accessible through the returned instance.

stride — The distance to the next element in mulitple of element count, where a value of 1 means the elements are adjacent in memory.

returns an object that implements **javax.realtime.device.RawFloat** and supports access to the specified range in the memory region.

Throws IllegalArgumentException when base is negative, count is not greater than zero, or stride is not greater than zero.

Throws SecurityException when the caller does not have permissions to access the given memory region or the specified range of addresses.

Throws OffsetOutOfBoundsException when base is invalid.

Throws SizeOutOfBoundsException when the memory addressed by the object would extend into an invalid range of memory.

Throws MemoryTypeConflictException when base does not point to memory that matches the type served by this factory.

Since
RTSJ 2.0

```
@SCJAllowed
@SCJMayAllocate({javax.safetycritical.annotate.AllocationContext.CURRENT})
@SCJMaySelfSuspend(false)
@SCJPhase({
    javax.safetycritical.annotate.Phase.STARTUP,
    javax.safetycritical.annotate.Phase.INITIALIZATION,
    javax.safetycritical.annotate.Phase.RUN,
    javax.safetycritical.annotate.Phase.CLEANUP })
public javax.realtime.device.RawFloatReader createRawFloatReader(long base,
  int count,
  int stride)
  throws
      java.lang.SecurityException, javax.realtime.OffsetOutOfBoundsException,
      javax.realtime.SizeOutOfBoundsException,
      javax.realtime.UnsupportedRawMemoryRegionException,
      javax.realtime.MemoryTypeConflictException
```

Create an instance of a class that implements **javax.realtime.device.RawFloatReader** and accesses memory of **javax.realtime.device.RawMemoryRegionFactorygetRegion** in the address range described by base, stride, and count. The actual extent of the memory addressed by the object is stride x *size of RawFloatReader* x count. The object is allocated in the current memory area of the calling thread.

base — The starting physical address accessible through the returned instance.

count — The number of memory elements accessible through the returned instance.

stride — The distance to the next element in mulitple of element count, where a value of 1 means the elements are adjacent in memory.

returns an object that implements **javax.realtime.device.RawFloatReader** and supports access to the specified range in the memory region.

Throws IllegalArgumentException when base is negative, count is not greater than zero, or stride is not greater than zero.

Throws SecurityException when the caller does not have permissions to access the given memory region or the specified range of addresses.

Throws OffsetOutOfBoundsException when base is invalid.

Throws SizeOutOfBoundsException when the memory addressed by the object would extend into an invalid range of memory.

Throws MemoryTypeConflictException when base does not point to memory that matches the type served by this factory.

Since
RTSJ 2.0

```
@SCJAllowed
@SCJMayAllocate({javax.safetycritical.annotate.AllocationContext.CURRENT})
@SCJMaySelfSuspend(false)
@SCJPhase({
    javax.safetycritical.annotate.Phase.STARTUP,
    javax.safetycritical.annotate.Phase.INITIALIZATION,
    javax.safetycritical.annotate.Phase.RUN,
    javax.safetycritical.annotate.Phase.CLEANUP })
public javax.realtime.device.RawFloatWriter createRawFloatWriter(long base,
  int count,
  int stride)
  throws
      java.lang.SecurityException, javax.realtime.OffsetOutOfBoundsException,
      javax.realtime.SizeOutOfBoundsException,
      javax.realtime.UnsupportedRawMemoryRegionException,
      javax.realtime.MemoryTypeConflictException
```

Create an instance of a class that implements **javax.realtime.device.RawFloatWriter** and accesses memory of **javax.realtime.device.RawMemoryRegionFactorygetRegion** in the address range described by base, stride, and count. The actual extent of the memory addressed by the object is stride x *size of RawFloatWriter* x count. The object is allocated in the current memory area of the calling thread.

base — The starting physical address accessible through the returned instance.

count — The number of memory elements accessible through the returned instance.

stride — The distance to the next element in mulitple of element count, where a value of 1 means the elements are adjacent in memory.

returns an object that implements **javax.realtime.device.RawFloatWriter** and supports access to the specified range in the memory region.

Throws IllegalArgumentException when base is negative, count is not greater than zero, or stride is not greater than zero.

Throws SecurityException when the caller does not have permissions to access the given memory region or the specified range of addresses.

Throws OffsetOutOfBoundsException when base is invalid.

Throws SizeOutOfBoundsException when the memory addressed by the object would extend into an invalid range of memory.

Throws MemoryTypeConflictException when base does not point to memory that matches the type served by this factory.

Since
RTSJ 2.0

```
@SCJAllowed
@SCJMayAllocate({javax.safetycritical.annotate.AllocationContext.CURRENT})
@SCJMaySelfSuspend(false)
@SCJPhase({
    javax.safetycritical.annotate.Phase.STARTUP,
    javax.safetycritical.annotate.Phase.INITIALIZATION,
    javax.safetycritical.annotate.Phase.RUN,
    javax.safetycritical.annotate.Phase.CLEANUP })
public javax.realtime.device.RawInt createRawInt(long base, int count, int stride)
  throws
      java.lang.SecurityException, javax.realtime.OffsetOutOfBoundsException,
      javax.realtime.SizeOutOfBoundsException,
      javax.realtime.UnsupportedRawMemoryRegionException,
      javax.realtime.MemoryTypeConflictException
```

Create an instance of a class that implements **javax.realtime.device.RawInt** and accesses memory of **javax.realtime.device.RawMemoryRegionFactoryget-Region** in the address range described by base, stride, and count. The actual extent of the memory addressed by the object is stride x *size of RawInt* x count. The object is allocated in the current memory area of the calling thread.

base — The starting physical address accessible through the returned instance.

count — The number of memory elements accessible through the returned instance.

stride — The distance to the next element in mulitple of element count, where a value of 1 means the elements are adjacent in memory.

returns an object that implements **javax.realtime.device.RawInt** and supports access to the specified range in the memory region.

Throws IllegalArgumentException when base is negative, count is not greater than zero, or stride is not greater than zero.

Throws SecurityException when the caller does not have permissions to access the given memory region or the specified range of addresses.

Throws OffsetOutOfBoundsException when base is invalid.

Throws SizeOutOfBoundsException when the memory addressed by the object would extend into an invalid range of memory.

Throws MemoryTypeConflictException when base does not point to memory that matches the type served by this factory.

Since
RTSJ 2.0

```
@SCJAllowed
@SCJMayAllocate({javax.safetycritical.annotate.AllocationContext.CURRENT})
@SCJMaySelfSuspend(false)
@SCJPhase({
    javax.safetycritical.annotate.Phase.STARTUP,
    javax.safetycritical.annotate.Phase.INITIALIZATION,
    javax.safetycritical.annotate.Phase.RUN,
    javax.safetycritical.annotate.Phase.CLEANUP })
public javax.realtime.device.RawIntReader createRawIntReader(long base,
  int count,
  int stride)
  throws
      java.lang.SecurityException, javax.realtime.OffsetOutOfBoundsException,
      javax.realtime.SizeOutOfBoundsException,
      javax.realtime.UnsupportedRawMemoryRegionException,
      javax.realtime.MemoryTypeConflictException
```

Create an instance of a class that implements **javax.realtime.device.RawIntReader** and accesses memory of **javax.realtime.device.RawMemoryRegionFactorygetRegion** in the address range described by base, stride, and count. The actual extent of the memory addressed by the object is stride x *size of RawIntReader* x count. The object is allocated in the current memory area of the calling thread.

base — The starting physical address accessible through the returned instance.

count — The number of memory elements accessible through the returned instance.

stride — The distance to the next element in mulitple of element count, where a value of 1 means the elements are adjacent in memory.

returns an object that implements **javax.realtime.device.RawIntReader** and supports access to the specified range in the memory region.

Throws IllegalArgumentException when base is negative, count is not greater than zero, or stride is not greater than zero.

Throws SecurityException when the caller does not have permissions to access the given memory region or the specified range of addresses.

Throws OffsetOutOfBoundsException when base is invalid.

Throws SizeOutOfBoundsException when the memory addressed by the object would extend into an invalid range of memory.

Throws MemoryTypeConflictException when base does not point to memory that matches the type served by this factory.

Since
RTSJ 2.0

```
@SCJAllowed
@SCJMayAllocate({javax.safetycritical.annotate.AllocationContext.CURRENT})
@SCJMaySelfSuspend(false)
@SCJPhase({
    javax.safetycritical.annotate.Phase.STARTUP,
    javax.safetycritical.annotate.Phase.INITIALIZATION,
    javax.safetycritical.annotate.Phase.RUN,
    javax.safetycritical.annotate.Phase.CLEANUP })
public javax.realtime.device.RawIntWriter createRawIntWriter(long base,
  int count,
  int stride)
  throws
      java.lang.SecurityException, javax.realtime.OffsetOutOfBoundsException,
      javax.realtime.SizeOutOfBoundsException,
      javax.realtime.UnsupportedRawMemoryRegionException,
      javax.realtime.MemoryTypeConflictException
```

Create an instance of a class that implements **javax.realtime.device.RawIntWriter** and accesses memory of **javax.realtime.device.RawMemoryRegionFactorygetRegion** in the address range described by base, stride, and count. The actual extent of the memory addressed by the object is stride x *size of RawIntWriter* x count. The object is allocated in the current memory area of the calling thread.

base — The starting physical address accessible through the returned instance.

count — The number of memory elements accessible through the returned instance.

stride — The distance to the next element in mulitple of element count, where a value of 1 means the elements are adjacent in memory.

returns an object that implements **javax.realtime.device.RawIntWriter** and supports access to the specified range in the memory region.

Throws IllegalArgumentException when base is negative, count is not greater than zero, or stride is not greater than zero.

Throws SecurityException when the caller does not have permissions to access the given memory region or the specified range of addresses.

Throws OffsetOutOfBoundsException when base is invalid.

Throws SizeOutOfBoundsException when the memory addressed by the object would extend into an invalid range of memory.

Throws MemoryTypeConflictException when base does not point to memory that matches the type served by this factory.

Since
RTSJ 2.0

```
@SCJAllowed
@SCJMayAllocate({javax.safetycritical.annotate.AllocationContext.CURRENT})
@SCJMaySelfSuspend(false)
@SCJPhase({
    javax.safetycritical.annotate.Phase.STARTUP,
    javax.safetycritical.annotate.Phase.INITIALIZATION,
    javax.safetycritical.annotate.Phase.RUN,
    javax.safetycritical.annotate.Phase.CLEANUP })
public javax.realtime.device.RawLong createRawLong(long base,
  int count,
  int stride)
  throws
      java.lang.SecurityException, javax.realtime.OffsetOutOfBoundsException,
      javax.realtime.SizeOutOfBoundsException,
      javax.realtime.UnsupportedRawMemoryRegionException,
      javax.realtime.MemoryTypeConflictException
```

Create an instance of a class that implements **javax.realtime.device.RawLong** and accesses memory of **javax.realtime.device.RawMemoryRegionFactoryget-Region** in the address range described by base, stride, and count. The actual extent of the memory addressed by the object is stride x *size of RawLong* x count. The object is allocated in the current memory area of the calling thread.

base — The starting physical address accessible through the returned instance.

count — The number of memory elements accessible through the returned instance.

stride — The distance to the next element in mulitple of element count, where a value of 1 means the elements are adjacent in memory.

returns an object that implements **javax.realtime.device.RawLong** and supports access to the specified range in the memory region.

Throws IllegalArgumentException when base is negative, count is not greater than zero, or stride is not greater than zero.

Throws SecurityException when the caller does not have permissions to access the given memory region or the specified range of addresses.

Throws OffsetOutOfBoundsException when base is invalid.

Throws SizeOutOfBoundsException when the memory addressed by the object would extend into an invalid range of memory.

Throws MemoryTypeConflictException when base does not point to memory that matches the type served by this factory.

Since
RTSJ 2.0

```
@SCJAllowed
@SCJMayAllocate({javax.safetycritical.annotate.AllocationContext.CURRENT})
@SCJMaySelfSuspend(false)
@SCJPhase({
    javax.safetycritical.annotate.Phase.STARTUP,
    javax.safetycritical.annotate.Phase.INITIALIZATION,
    javax.safetycritical.annotate.Phase.RUN,
    javax.safetycritical.annotate.Phase.CLEANUP })
public javax.realtime.device.RawLongReader createRawLongReader(long base,
  int count,
  int stride)
  throws
      java.lang.SecurityException, javax.realtime.OffsetOutOfBoundsException,
      javax.realtime.SizeOutOfBoundsException,
      javax.realtime.UnsupportedRawMemoryRegionException,
      javax.realtime.MemoryTypeConflictException
```

Create an instance of a class that implements **javax.realtime.device.RawLongReader** and accesses memory of **javax.realtime.device.RawMemoryRegionFactorygetRegion** in the address range described by base, stride, and count. The actual extent of the memory addressed by the object is stride x *size of RawLongReader* x count. The object is allocated in the current memory area of the calling thread.

base — The starting physical address accessible through the returned instance.

count — The number of memory elements accessible through the returned instance.

stride — The distance to the next element in mulitple of element count, where a value of 1 means the elements are adjacent in memory.

returns an object that implements **javax.realtime.device.RawLongReader** and supports access to the specified range in the memory region.

Throws IllegalArgumentException when base is negative, count is not greater than zero, or stride is not greater than zero.

Throws SecurityException when the caller does not have permissions to access the given memory region or the specified range of addresses.

Throws OffsetOutOfBoundsException when base is invalid.

Throws SizeOutOfBoundsException when the memory addressed by the object would extend into an invalid range of memory.

Throws MemoryTypeConflictException when base does not point to memory that matches the type served by this factory.

Since
RTSJ 2.0

```
@SCJAllowed
@SCJMayAllocate({javax.safetycritical.annotate.AllocationContext.CURRENT})
@SCJMaySelfSuspend(false)
@SCJPhase({
    javax.safetycritical.annotate.Phase.STARTUP,
    javax.safetycritical.annotate.Phase.INITIALIZATION,
    javax.safetycritical.annotate.Phase.RUN,
    javax.safetycritical.annotate.Phase.CLEANUP })
public javax.realtime.device.RawLongWriter createRawLongWriter(long base,
  int count,
  int stride)
  throws
      java.lang.SecurityException, javax.realtime.OffsetOutOfBoundsException,
      javax.realtime.SizeOutOfBoundsException,
      javax.realtime.UnsupportedRawMemoryRegionException,
      javax.realtime.MemoryTypeConflictException
```

Create an instance of a class that implements **javax.realtime.device.RawLongWriter** and accesses memory of **javax.realtime.device.RawMemoryRegionFactorygetRegion** in the address range described by base, stride, and count. The actual extent of the memory addressed by the object is stride x *size of RawLongWriter* x count. The object is allocated in the current memory area of the calling thread.

base — The starting physical address accessible through the returned instance.

count — The number of memory elements accessible through the returned instance.

stride — The distance to the next element in mulitple of element count, where a value of 1 means the elements are adjacent in memory.

returns an object that implements **javax.realtime.device.RawLongWriter** and supports access to the specified range in the memory region.

Throws IllegalArgumentException when base is negative, count is not greater than zero, or stride is not greater than zero.

Throws SecurityException when the caller does not have permissions to access the given memory region or the specified range of addresses.

Throws OffsetOutOfBoundsException when base is invalid.

Throws SizeOutOfBoundsException when the memory addressed by the object would extend into an invalid range of memory.

Throws MemoryTypeConflictException when base does not point to memory that matches the type served by this factory.

Since
RTSJ 2.0

```
@SCJAllowed
@SCJMayAllocate({javax.safetycritical.annotate.AllocationContext.CURRENT})
@SCJMaySelfSuspend(false)
@SCJPhase({
    javax.safetycritical.annotate.Phase.STARTUP,
    javax.safetycritical.annotate.Phase.INITIALIZATION,
    javax.safetycritical.annotate.Phase.RUN,
    javax.safetycritical.annotate.Phase.CLEANUP })
public javax.realtime.device.RawShort createRawShort(long base,
  int count,
  int stride)
  throws
      java.lang.SecurityException, javax.realtime.OffsetOutOfBoundsException,
      javax.realtime.SizeOutOfBoundsException,
      javax.realtime.UnsupportedRawMemoryRegionException,
      javax.realtime.MemoryTypeConflictException
```

Create an instance of a class that implements **javax.realtime.device.RawShort** and accesses memory of **javax.realtime.device.RawMemoryRegionFactoryget-Region** in the address range described by base, stride, and count. The actual extent of the memory addressed by the object is stride x *size of RawShort* x count. The object is allocated in the current memory area of the calling thread.

base — The starting physical address accessible through the returned instance.

count — The number of memory elements accessible through the returned instance.

stride — The distance to the next element in mulitple of element count, where a value of 1 means the elements are adjacent in memory.

returns an object that implements **javax.realtime.device.RawShort** and supports access to the specified range in the memory region.

Throws IllegalArgumentException when base is negative, count is not greater than zero, or stride is not greater than zero.

Throws SecurityException when the caller does not have permissions to access the given memory region or the specified range of addresses.

Throws OffsetOutOfBoundsException when base is invalid.

Throws SizeOutOfBoundsException when the memory addressed by the object would extend into an invalid range of memory.

Throws MemoryTypeConflictException when base does not point to memory that matches the type served by this factory.

Since
RTSJ 2.0

```
@SCJAllowed
@SCJMayAllocate({javax.safetycritical.annotate.AllocationContext.CURRENT})
@SCJMaySelfSuspend(false)
@SCJPhase({
    javax.safetycritical.annotate.Phase.STARTUP,
    javax.safetycritical.annotate.Phase.INITIALIZATION,
    javax.safetycritical.annotate.Phase.RUN,
    javax.safetycritical.annotate.Phase.CLEANUP })
public javax.realtime.device.RawShortReader createRawShortReader(long base,
  int count,
  int stride)
  throws
      java.lang.SecurityException, javax.realtime.OffsetOutOfBoundsException,
      javax.realtime.SizeOutOfBoundsException,
      javax.realtime.UnsupportedRawMemoryRegionException,
      javax.realtime.MemoryTypeConflictException
```

Create an instance of a class that implements **javax.realtime.device.RawShortReader** and accesses memory of **javax.realtime.device.RawMemoryRegionFactorygetRegion** in the address range described by base, stride, and count. The actual extent of the memory addressed by the object is stride x *size of RawShortReader* x count. The object is allocated in the current memory area of the calling thread.

base — The starting physical address accessible through the returned instance.

count — The number of memory elements accessible through the returned instance.

stride — The distance to the next element in mulitple of element count, where a value of 1 means the elements are adjacent in memory.

returns an object that implements **javax.realtime.device.RawShortReader** and supports access to the specified range in the memory region.

Throws IllegalArgumentException when base is negative, count is not greater than zero, or stride is not greater than zero.

Throws SecurityException when the caller does not have permissions to access the given memory region or the specified range of addresses.

Throws OffsetOutOfBoundsException when base is invalid.

Throws SizeOutOfBoundsException when the memory addressed by the object would extend into an invalid range of memory.

Throws MemoryTypeConflictException when base does not point to memory that matches the type served by this factory.

Since
RTSJ 2.0

```
@SCJAllowed
@SCJMayAllocate({javax.safetycritical.annotate.AllocationContext.CURRENT})
@SCJMaySelfSuspend(false)
@SCJPhase({
    javax.safetycritical.annotate.Phase.STARTUP,
    javax.safetycritical.annotate.Phase.INITIALIZATION,
    javax.safetycritical.annotate.Phase.RUN,
    javax.safetycritical.annotate.Phase.CLEANUP })
public javax.realtime.device.RawShortWriter createRawShortWriter(long base,
  int count,
  int stride)
  throws
     java.lang.SecurityException, javax.realtime.OffsetOutOfBoundsException,
     javax.realtime.SizeOutOfBoundsException,
     javax.realtime.UnsupportedRawMemoryRegionException,
     javax.realtime.MemoryTypeConflictException
```

Create an instance of a class that implements **javax.realtime.device.RawShortWriter** and accesses memory of **javax.realtime.device.RawMemoryRegionFactorygetRegion** in the address range described by base, stride, and count. The actual extent of the memory addressed by the object is stride x *size of RawShortWriter* x count. The object is allocated in the current memory area of the calling thread.

base — The starting physical address accessible through the returned instance.

count — The number of memory elements accessible through the returned instance.

stride — The distance to the next element in mulitple of element count, where a value of 1 means the elements are adjacent in memory.

returns an object that implements **javax.realtime.device.RawShortWriter** and supports access to the specified range in the memory region.

Throws IllegalArgumentException when base is negative, count is not greater than zero, or stride is not greater than zero.

Throws SecurityException when the caller does not have permissions to access the given memory region or the specified range of addresses.

Throws OffsetOutOfBoundsException when base is invalid.

Throws SizeOutOfBoundsException when the memory addressed by the object would extend into an invalid range of memory.

Throws MemoryTypeConflictException when base does not point to memory that matches the type served by this factory.

Since
RTSJ 2.0

```
@SCJAllowed
@SCJMayAllocate({})
@SCJMaySelfSuspend(false)
@SCJPhase({
    javax.safetycritical.annotate.Phase.STARTUP,
    javax.safetycritical.annotate.Phase.INITIALIZATION,
    javax.safetycritical.annotate.Phase.RUN,
    javax.safetycritical.annotate.Phase.CLEANUP })
public java.lang.String getName( )
```

Determine the name of the region for which this factory creates raw memory objects.

returns the name of the region of this factory.

```
@SCJAllowed
@SCJMayAllocate({})
@SCJMaySelfSuspend(false)
@SCJPhase({
    javax.safetycritical.annotate.Phase.STARTUP,
    javax.safetycritical.annotate.Phase.INITIALIZATION,
    javax.safetycritical.annotate.Phase.RUN,
    javax.safetycritical.annotate.Phase.CLEANUP })
public javax.realtime.device.RawMemoryRegion getRegion( )
```

Determine for what region this factory creates raw memory objects.

returns the region of this factory.

5.2.18 javax.realtime.device.RawMemoryFactory

Declaration

```
@SCJAllowed
 public class RawMemoryFactory extends java.lang.Object
```

Since
RTSJ 2.0

Description

This class is the hub of a system that constructs special purpose objects to access particular types and ranges of raw memory. This facility is supported by the RawMemoryRegionFactory methods. An application developer can use this method to add support for additional memory regions.

Each create method returns an object of the corresponding type, e.g., the createRawByte(RawMemoryRegion, long, int, int)} method returns a reference to an object that implements the **javax.realtime.device.RawByte** interface and supports access to the requested type of memory and address range. Each create method is permitted to optimize error checking and access based on the requested memory type and address range.

The usage pattern for raw memory, assuming the necessary factory has been registered, is illustrated by this example.

```
// Get an accessor object that can access memory starting at
// baseAddress, for size bytes.
RawInt memory =
  RawMemoryFactory.createRawInt(RawMemoryFactory.MEMORY_MAPPED_REGION,
                                address, count, stride, false);
// Use the accessor to load from and store to raw memory.
int loadedData = memory.getInt(someOffset);
memory.setInt(otherOffset, intVal);
```

When an application needs to access a class of memory that is not already supported by a registered factory, the developer must implement and register a factory that implements the **javax.realtime.device.RawMemoryRegionFactory**) which can create objects to access memory in that region.

A raw memory region factory is identified by a **javax.realtime.device.RawMemoryRegion** that is used by each create method, e.g., **createRawByte(RawMemoryRegion, long, int, int)** , to locate the appropriate factory. The name is provided to **register(RawMemoryRegionFactory)** through the factory's **javax-.realtime.device.RawMemoryRegionFactorygetName** method.

The **register(RawMemoryRegionFactory)** method is only used when by application code when it needs to add support for a new type of raw memory.

Whether a give offset addresses a high-order or low-order byte of an aligned short in memory is determined by the value of the javax.realtime.RealtimeSystem.BYTE_ORDER static byte variable in class javax.realtime.RealtimeSystem, the start address of the object, and the offset given the stride of the object. Regardless of the byte ordering, accessor methods continue to select bytes starting at offset from the base address and continuing toward greater addresses.

A raw memory region cannot contain references to Java objects. Such a capability would be unsafe (since it could be used to defeat Java's type checking) and error prone (since it is sensitive to the specific representational choices made by the Java compiler).

Atomic loads and stores on raw memory are defined in terms of physical memory. This memory may be accessible to threads outside the JVM and to non-programmed access (e.g., DMA). Consequently, atomic access must be sup-

ported by hardware. This specification is written with the assumption that all suitable hardware platforms support atomic loads from raw memory for aligned bytes, shorts, and ints. Atomic access beyond the specified minimum may be supported by the implementation.

Storing values into raw memory is more hardware-dependent than loading values. Many processor architectures do not support atomic stores of variables except for aligned stores of the processor's word size. For instance, storing a byte into memory might require reading a 32-bit quantity into a processor register, updating the register to reflect the new byte value, then restoring the whole 32-bit quantity. Changes to other bytes in the 32-bit quantity that take place between the load and the store are lost.

Some processors have mechanisms that can be used to implement an atomic store of a byte, but those mechanisms are often slow and not universally supported.

This class need not support unaligned access to data; but if it does, it is not require the implementation to make such access atomic. Accesses to data aligned on its natural boundary will be atomic if the processor implements atomic loads and stores of that data size.

Except where noted, accesses to raw memory are not atomic with respect to the memory or with respect to schedulable objects. A raw memory region could be updated by another schedulable object, or even unmapped in the middle of an access method, or even *removed* mid method.

The characteristics of raw-memory access are necessarily platform dependent. This specification provides a minimum requirement for the SCJ platform, but it also supports optional system properties that identify a platform's level of support for atomic raw put and get. The properties represent a four-dimensional sparse array of access type, data type, alignment, and atomicity with boolean values indicating whether that combination of access attributes is atomic. The default value for array entries is false. The permissable values of these array entries are:

- Access type - possible values are *read* and *write*.
- Data type - possible values are *byte*, *short*, *int*, *long*, *float*, and *double*.
- Alignment - possible values are 0 through 7, inclusive. For each data type, the possible alignments range from:
  - 0 means aligned
  - 1 to (data size-1) means only the first byte of the data is alignment bytes away from natural alignment
- Atomicity - possible values are *processor*, *smp*, and *memory*.
  - *processor* means that access is atomic with respect to other schedulable objects on that processor.

- *smp* means that access is processor atomic, and atomic across the processors in an SMP.
- *memory* means that access is SMP atomic, and atomic with respect to all access to the memory, including DMA hardware.

The true values in the table are represented by properties of the following form. javax.realtime.atomicaccess_<access>_<type>_<alignment>_atomicity=true for example,

javax.realtime.atomicaccess_read_byte_0_memory=true

Table entries with a value of false may be explicitly represented, but since false is the default value, such properties are redundant.

All raw memory access is treated as volatile, and *serialized*. The infrastructure must be forced to read memory or write to memory on each call to a raw memory objects's getter or setter method, and to complete the reads and writes in the order they appear in the program order.

Fields

@SCJAllowed
public static final javax.realtime.device.RawMemoryRegion IO_PORT_MAPPED_REGION

This raw memory region is predefined for access to I/O device space implemented by processor instructions, such as the x86 in and out instructions.

@SCJAllowed
public static final javax.realtime.device.RawMemoryRegion MEMORY_MAPPED_REGION

This raw memory region is predefined for request access to memory mapped I/O devices.

Constructors

```
@SCJAllowed
@SCJMaySelfSuspend(false)
@SCJPhase({
   javax.safetycritical.annotate.Phase.STARTUP,
   javax.safetycritical.annotate.Phase.INITIALIZATION})
@SCJMayAllocate({
   javax.safetycritical.annotate.AllocationContext.CURRENT,
   javax.safetycritical.annotate.AllocationContext.INNER,
   javax.safetycritical.annotate.AllocationContext.OUTER})
public RawMemoryFactory( )
```

Create an empty factory. For a factory with support for the platform defined regions, use **javax.realtime.device.RawMemoryFactorygetDefaultFactory** instead.

Methods

```
@SCJAllowed
@SCJMayAllocate({javax.safetycritical.annotate.AllocationContext.CURRENT})
@SCJMaySelfSuspend(false)
@SCJPhase({
    javax.safetycritical.annotate.Phase.STARTUP,
    javax.safetycritical.annotate.Phase.INITIALIZATION,
    javax.safetycritical.annotate.Phase.RUN,
    javax.safetycritical.annotate.Phase.CLEANUP })
public javax.realtime.device.RawByte createRawByte(RawMemoryRegion region,
  long base,
  int count,
  int stride)
  throws
      java.lang.SecurityException, javax.realtime.OffsetOutOfBoundsException,
      javax.realtime.SizeOutOfBoundsException,
      javax.realtime.MemoryTypeConflictException,
      javax.realtime.UnsupportedRawMemoryRegionException
```

Create an instance of a class that implements **javax.realtime.device.RawByte** and accesses memory of region in the address range described by base, stride, and count. The actual extent of the memory addressed by the object is stride x *size of RawByte* x count. The object is allocated in the current memory area of the calling thread.

region — The address space from which the new instance should be taken.

base — The starting physical address accessible through the returned instance.

count — The number of memory elements accessible through the returned instance.

stride — The distance to the next element as a mulitple of element size, where 1 means the elements are adjacent in memory.

returns an object that implements **javax.realtime.device.RawByte** and supports access to the specified range in the memory region.

Throws IllegalArgumentException when base is negative, count is not greater than zero, or stride is less than one.

Throws SecurityException when the caller does not have permissions to access the given memory region or the specified range of addresses.

Throws OffsetOutOfBoundsException when base is invalid.

Throws SizeOutOfBoundsException when the memory addressed by the object would extend into an invalid range of memory.

Throws MemoryTypeConflictException when base does not point to a memory that matches the type served by this factory.

```
@SCJAllowed
@SCJMayAllocate({javax.safetycritical.annotate.AllocationContext.CURRENT})
@SCJMaySelfSuspend(false)
@SCJPhase({
    javax.safetycritical.annotate.Phase.STARTUP,
    javax.safetycritical.annotate.Phase.INITIALIZATION,
    javax.safetycritical.annotate.Phase.RUN,
    javax.safetycritical.annotate.Phase.CLEANUP })
public javax.realtime.device.RawByteReader createRawByteReader(
  RawMemoryRegion region,
  long base,
  int count,
  int stride)
  throws
      java.lang.SecurityException, javax.realtime.OffsetOutOfBoundsException,
      javax.realtime.SizeOutOfBoundsException,
      javax.realtime.MemoryTypeConflictException,
      javax.realtime.UnsupportedRawMemoryRegionException
```

Create an instance of a class that implements **javax.realtime.device.RawByteReader** and accesses memory of region in the address range described by base, stride, and count. The actual extent of the memory addressed by the object is stride x *size of RawByteReader* x count. The object is allocated in the current memory area of the calling thread.

region — The address space from which the new instance should be taken.

base — The starting physical address accessible through the returned instance.

count — The number of memory elements accessible through the returned instance.

stride — The distance to the next element as a mulitple of element size, where 1 means the elements are adjacent in memory.

returns an object that implements **javax.realtime.device.RawByteReader** and supports access to the specified range in the memory region.

Throws IllegalArgumentException when base is negative, count is not greater than zero, or stride is less than one.

Throws SecurityException when the caller does not have permissions to access the given memory region or the specified range of addresses.

Throws OffsetOutOfBoundsException when base is invalid.

Throws SizeOutOfBoundsException when the memory addressed by the object would extend into an invalid range of memory.

Throws MemoryTypeConflictException when base does not point to a memory that matches the type served by this factory.

```
@SCJAllowed
@SCJMayAllocate({javax.safetycritical.annotate.AllocationContext.CURRENT})
@SCJMaySelfSuspend(false)
@SCJPhase({
    javax.safetycritical.annotate.Phase.STARTUP,
    javax.safetycritical.annotate.Phase.INITIALIZATION,
    javax.safetycritical.annotate.Phase.RUN,
    javax.safetycritical.annotate.Phase.CLEANUP })
public javax.realtime.device.RawByteWriter createRawByteWriter(
  RawMemoryRegion region,
  long base,
  int count,
  int stride)
  throws
      java.lang.SecurityException, javax.realtime.OffsetOutOfBoundsException,
      javax.realtime.SizeOutOfBoundsException,
      javax.realtime.MemoryTypeConflictException,
      javax.realtime.UnsupportedRawMemoryRegionException
```

Create an instance of a class that implements **javax.realtime.device.RawByteWriter** and accesses memory of region in the address range described by base, stride, and count. The actual extent of the memory addressed by the object is stride x *size of RawByteWriter* x count. The object is allocated in the current memory area of the calling thread.

region — The address space from which the new instance should be taken.

base — The starting physical address accessible through the returned instance.

count — The number of memory elements accessible through the returned instance.

stride — The distance to the next element as a mulitple of element size, where 1 means the elements are adjacent in memory.

returns an object that implements **javax.realtime.device.RawByteWriter** and supports access to the specified range in the memory region.

Throws IllegalArgumentException when base is negative, count is not greater than zero, or stride is less than one.

Throws SecurityException when the caller does not have permissions to access the given memory region or the specified range of addresses.

Throws OffsetOutOfBoundsException when base is invalid.

Throws SizeOutOfBoundsException when the memory addressed by the object would extend into an invalid range of memory.

Throws MemoryTypeConflictException when base does not point to a memory that matches the type served by this factory.

```
@SCJAllowed
@SCJMayAllocate({javax.safetycritical.annotate.AllocationContext.CURRENT})
@SCJMaySelfSuspend(false)
@SCJPhase({
    javax.safetycritical.annotate.Phase.STARTUP,
    javax.safetycritical.annotate.Phase.INITIALIZATION,
    javax.safetycritical.annotate.Phase.RUN,
    javax.safetycritical.annotate.Phase.CLEANUP })
public javax.realtime.device.RawDouble createRawDouble(RawMemoryRegion region,
  long base,
  int count,
  int stride)
  throws
      java.lang.SecurityException, javax.realtime.OffsetOutOfBoundsException,
      javax.realtime.SizeOutOfBoundsException,
      javax.realtime.MemoryTypeConflictException,
      javax.realtime.UnsupportedRawMemoryRegionException
```

Create an instance of a class that implements **javax.realtime.device.RawDouble** and accesses memory of region in the address range described by base, stride, and count. The actual extent of the memory addressed by the object is stride x *size of RawDouble* x count. The object is allocated in the current memory area of the calling thread.

region — The address space from which the new instance should be taken.

base — The starting physical address accessible through the returned instance.

count — The number of memory elements accessible through the returned instance.

stride — The distance to the next element as a mulitple of element size, where 1 means the elements are adjacent in memory.

returns an object that implements **javax.realtime.device.RawDouble** and supports access to the specified range in the memory region.

Throws IllegalArgumentException when base is negative, count is not greater than zero, or stride is less than one.

Throws SecurityException when the caller does not have permissions to access the given memory region or the specified range of addresses.

Throws OffsetOutOfBoundsException when base is invalid.

Throws SizeOutOfBoundsException when the memory addressed by the object would extend into an invalid range of memory.

Throws MemoryTypeConflictException when base does not point to a memory that matches the type served by this factory.

```
@SCJAllowed
@SCJMayAllocate({javax.safetycritical.annotate.AllocationContext.CURRENT})
@SCJMaySelfSuspend(false)
@SCJPhase({
    javax.safetycritical.annotate.Phase.STARTUP,
    javax.safetycritical.annotate.Phase.INITIALIZATION,
    javax.safetycritical.annotate.Phase.RUN,
    javax.safetycritical.annotate.Phase.CLEANUP })
public javax.realtime.device.RawDoubleReader createRawDoubleReader(
  RawMemoryRegion region,
  long base,
  int count,
  int stride)
  throws
      java.lang.SecurityException, javax.realtime.OffsetOutOfBoundsException,
      javax.realtime.SizeOutOfBoundsException,
      javax.realtime.MemoryTypeConflictException,
      javax.realtime.UnsupportedRawMemoryRegionException
```

Create an instance of a class that implements **javax.realtime.device.RawDoubleReader** and accesses memory of region in the address range described by base, stride, and count. The actual extent of the memory addressed by the object is stride x *size of RawDoubleReader* x count. The object is allocated in the current memory area of the calling thread.

region — The address space from which the new instance should be taken.

base — The starting physical address accessible through the returned instance.

count — The number of memory elements accessible through the returned instance.

stride — The distance to the next element as a mulitple of element size, where 1 means the elements are adjacent in memory.

returns an object that implements **javax.realtime.device.RawDoubleReader** and supports access to the specified range in the memory region.

Throws IllegalArgumentException when base is negative, count is not greater than zero, or stride is less than one.

Throws SecurityException when the caller does not have permissions to access the given memory region or the specified range of addresses.

Throws OffsetOutOfBoundsException when base is invalid.

Throws SizeOutOfBoundsException when the memory addressed by the object would extend into an invalid range of memory.

Throws MemoryTypeConflictException when base does not point to a memory that matches the type served by this factory.

```
@SCJAllowed
@SCJMayAllocate({javax.safetycritical.annotate.AllocationContext.CURRENT})
@SCJMaySelfSuspend(false)
@SCJPhase({
    javax.safetycritical.annotate.Phase.STARTUP,
    javax.safetycritical.annotate.Phase.INITIALIZATION,
    javax.safetycritical.annotate.Phase.RUN,
    javax.safetycritical.annotate.Phase.CLEANUP })
public javax.realtime.device.RawDoubleWriter createRawDoubleWriter(
  RawMemoryRegion region,
  long base,
  int count,
  int stride)
  throws
      java.lang.SecurityException, javax.realtime.OffsetOutOfBoundsException,
      javax.realtime.SizeOutOfBoundsException,
      javax.realtime.MemoryTypeConflictException,
      javax.realtime.UnsupportedRawMemoryRegionException
```

Create an instance of a class that implements **javax.realtime.device.RawDoubleWriter** and accesses memory of region in the address range described by base, stride, and count. The actual extent of the memory addressed by the object is stride x *size of RawDoubleWriter* x count. The object is allocated in the current memory area of the calling thread.

region — The address space from which the new instance should be taken.

base — The starting physical address accessible through the returned instance.

count — The number of memory elements accessible through the returned instance.

stride — The distance to the next element as a mulitple of element size, where 1 means the elements are adjacent in memory.

returns an object that implements **javax.realtime.device.RawDoubleWriter** and supports access to the specified range in the memory region.

Throws IllegalArgumentException when base is negative, count is not greater than zero, or stride is less than one.

Throws SecurityException when the caller does not have permissions to access the given memory region or the specified range of addresses.

Throws OffsetOutOfBoundsException when base is invalid.

Throws SizeOutOfBoundsException when the memory addressed by the object would extend into an invalid range of memory.

Throws MemoryTypeConflictException when base does not point to a memory that matches the type served by this factory.

```
@SCJAllowed
@SCJMayAllocate({javax.safetycritical.annotate.AllocationContext.CURRENT})
@SCJMaySelfSuspend(false)
@SCJPhase({
    javax.safetycritical.annotate.Phase.STARTUP,
    javax.safetycritical.annotate.Phase.INITIALIZATION,
    javax.safetycritical.annotate.Phase.RUN,
    javax.safetycritical.annotate.Phase.CLEANUP })
public javax.realtime.device.RawFloat createRawFloat(RawMemoryRegion region,
  long base,
  int count,
  int stride)
  throws
      java.lang.SecurityException, javax.realtime.OffsetOutOfBoundsException,
      javax.realtime.SizeOutOfBoundsException,
      javax.realtime.MemoryTypeConflictException,
      javax.realtime.UnsupportedRawMemoryRegionException
```

Create an instance of a class that implements **javax.realtime.device.RawFloat** and accesses memory of region in the address range described by base, stride, and count. The actual extent of the memory addressed by the object is stride x *size of RawFloat* x count. The object is allocated in the current memory area of the calling thread.

region — The address space from which the new instance should be taken.

base — The starting physical address accessible through the returned instance.

count — The number of memory elements accessible through the returned instance.

stride — The distance to the next element as a mulitple of element size, where 1 means the elements are adjacent in memory.

returns an object that implements **javax.realtime.device.RawFloat** and supports access to the specified range in the memory region.

Throws IllegalArgumentException when base is negative, count is not greater than zero, or stride is less than one.

Throws SecurityException when the caller does not have permissions to access the given memory region or the specified range of addresses.

Throws OffsetOutOfBoundsException when base is invalid.

Throws SizeOutOfBoundsException when the memory addressed by the object would extend into an invalid range of memory.

Throws MemoryTypeConflictException when base does not point to a memory that matches the type served by this factory.

```
@SCJAllowed
@SCJMayAllocate({javax.safetycritical.annotate.AllocationContext.CURRENT})
@SCJMaySelfSuspend(false)
@SCJPhase({
    javax.safetycritical.annotate.Phase.STARTUP,
    javax.safetycritical.annotate.Phase.INITIALIZATION,
    javax.safetycritical.annotate.Phase.RUN,
    javax.safetycritical.annotate.Phase.CLEANUP })
public javax.realtime.device.RawFloatReader createRawFloatReader(
  RawMemoryRegion region,
  long base,
  int count,
  int stride)
  throws
     java.lang.SecurityException, javax.realtime.OffsetOutOfBoundsException,
     javax.realtime.SizeOutOfBoundsException,
     javax.realtime.MemoryTypeConflictException,
     javax.realtime.UnsupportedRawMemoryRegionException
```

Create an instance of a class that implements **javax.realtime.device.RawFloatReader** and accesses memory of region in the address range described by base, stride, and count. The actual extent of the memory addressed by the object is stride x *size of RawFloatReader* x count. The object is allocated in the current memory area of the calling thread.

region — The address space from which the new instance should be taken.

base — The starting physical address accessible through the returned instance.

count — The number of memory elements accessible through the returned instance.

stride — The distance to the next element as a mulitple of element size, where 1 means the elements are adjacent in memory.

returns an object that implements **javax.realtime.device.RawFloatReader** and supports access to the specified range in the memory region.

Throws IllegalArgumentException when base is negative, count is not greater than zero, or stride is less than one.

Throws SecurityException when the caller does not have permissions to access the given memory region or the specified range of addresses.

Throws OffsetOutOfBoundsException when base is invalid.

Throws SizeOutOfBoundsException when the memory addressed by the object would extend into an invalid range of memory.

Throws MemoryTypeConflictException when base does not point to a memory that matches the type served by this factory.

```
@SCJAllowed
@SCJMayAllocate({javax.safetycritical.annotate.AllocationContext.CURRENT})
@SCJMaySelfSuspend(false)
@SCJPhase({
    javax.safetycritical.annotate.Phase.STARTUP,
    javax.safetycritical.annotate.Phase.INITIALIZATION,
    javax.safetycritical.annotate.Phase.RUN,
    javax.safetycritical.annotate.Phase.CLEANUP })
public javax.realtime.device.RawFloatWriter createRawFloatWriter(
  RawMemoryRegion region,
  long base,
  int count,
  int stride)
  throws
      java.lang.SecurityException, javax.realtime.OffsetOutOfBoundsException,
      javax.realtime.SizeOutOfBoundsException,
      javax.realtime.MemoryTypeConflictException,
      javax.realtime.UnsupportedRawMemoryRegionException
```

Create an instance of a class that implements **javax.realtime.device.RawFloatWriter** and accesses memory of region in the address range described by base, stride, and count. The actual extent of the memory addressed by the object is stride x *size of RawFloatWriter* x count. The object is allocated in the current memory area of the calling thread.

region — The address space from which the new instance should be taken.

base — The starting physical address accessible through the returned instance.

count — The number of memory elements accessible through the returned instance.

stride — The distance to the next element as a mulitple of element size, where 1 means the elements are adjacent in memory.

returns an object that implements **javax.realtime.device.RawFloatWriter** and supports access to the specified range in the memory region.

Throws IllegalArgumentException when base is negative, count is not greater than zero, or stride is less than one.

Throws SecurityException when the caller does not have permissions to access the given memory region or the specified range of addresses.

Throws OffsetOutOfBoundsException when base is invalid.

Throws SizeOutOfBoundsException when the memory addressed by the object would extend into an invalid range of memory.

Throws MemoryTypeConflictException when base does not point to a memory that matches the type served by this factory.

```
@SCJAllowed
@SCJMayAllocate({javax.safetycritical.annotate.AllocationContext.CURRENT})
@SCJMaySelfSuspend(false)
@SCJPhase({
    javax.safetycritical.annotate.Phase.STARTUP,
    javax.safetycritical.annotate.Phase.INITIALIZATION,
    javax.safetycritical.annotate.Phase.RUN,
    javax.safetycritical.annotate.Phase.CLEANUP })
public javax.realtime.device.RawInt createRawInt(RawMemoryRegion region,
  long base,
  int count,
  int stride)
  throws
      java.lang.SecurityException, javax.realtime.OffsetOutOfBoundsException,
      javax.realtime.SizeOutOfBoundsException,
      javax.realtime.MemoryTypeConflictException,
      javax.realtime.UnsupportedRawMemoryRegionException
```

Create an instance of a class that implements **javax.realtime.device.RawInt** and accesses memory of region in the address range described by base, stride, and count. The actual extent of the memory addressed by the object is stride x *size of RawInt* x count. The object is allocated in the current memory area of the calling thread.

region — The address space from which the new instance should be taken.

base — The starting physical address accessible through the returned instance.

count — The number of memory elements accessible through the returned instance.

stride — The distance to the next element as a mulitple of element size, where 1 means the elements are adjacent in memory.

returns an object that implements **javax.realtime.device.RawInt** and supports access to the specified range in the memory region.

Throws IllegalArgumentException when base is negative, count is not greater than zero, or stride is less than one.

Throws SecurityException when the caller does not have permissions to access the given memory region or the specified range of addresses.

Throws OffsetOutOfBoundsException when base is invalid.

Throws SizeOutOfBoundsException when the memory addressed by the object would extend into an invalid range of memory.

Throws MemoryTypeConflictException when base does not point to a memory that matches the type served by this factory.

```
@SCJAllowed
@SCJMayAllocate({javax.safetycritical.annotate.AllocationContext.CURRENT})
@SCJMaySelfSuspend(false)
@SCJPhase({
    javax.safetycritical.annotate.Phase.STARTUP,
    javax.safetycritical.annotate.Phase.INITIALIZATION,
    javax.safetycritical.annotate.Phase.RUN,
    javax.safetycritical.annotate.Phase.CLEANUP })
public javax.realtime.device.RawIntReader createRawIntReader(
  RawMemoryRegion region,
  long base,
  int count,
  int stride)
  throws
      java.lang.SecurityException, javax.realtime.OffsetOutOfBoundsException,
      javax.realtime.SizeOutOfBoundsException,
      javax.realtime.MemoryTypeConflictException,
      javax.realtime.UnsupportedRawMemoryRegionException
```

Create an instance of a class that implements **javax.realtime.device.RawIntReader** and accesses memory of region in the address range described by base, stride, and count. The actual extent of the memory addressed by the object is stride x *size of RawIntReader* x count. The object is allocated in the current memory area of the calling thread.

region — The address space from which the new instance should be taken.

base — The starting physical address accessible through the returned instance.

count — The number of memory elements accessible through the returned instance.

stride — The distance to the next element as a mulitple of element size, where 1 means the elements are adjacent in memory.

returns an object that implements **javax.realtime.device.RawIntReader** and supports access to the specified range in the memory region.

Throws IllegalArgumentException when base is negative, count is not greater than zero, or stride is less than one.

Throws SecurityException when the caller does not have permissions to access the given memory region or the specified range of addresses.

Throws OffsetOutOfBoundsException when base is invalid.

Throws SizeOutOfBoundsException when the memory addressed by the object would extend into an invalid range of memory.

Throws MemoryTypeConflictException when base does not point to a memory that matches the type served by this factory.

```
@SCJAllowed
@SCJMayAllocate({javax.safetycritical.annotate.AllocationContext.CURRENT})
@SCJMaySelfSuspend(false)
@SCJPhase({
    javax.safetycritical.annotate.Phase.STARTUP,
    javax.safetycritical.annotate.Phase.INITIALIZATION,
    javax.safetycritical.annotate.Phase.RUN,
    javax.safetycritical.annotate.Phase.CLEANUP })
public javax.realtime.device.RawIntWriter createRawIntWriter(
  RawMemoryRegion region,
  long base,
  int count,
  int stride)
  throws
      java.lang.SecurityException, javax.realtime.OffsetOutOfBoundsException,
      javax.realtime.SizeOutOfBoundsException,
      javax.realtime.MemoryTypeConflictException,
      javax.realtime.UnsupportedRawMemoryRegionException
```

Create an instance of a class that implements **javax.realtime.device.RawIntWriter** and accesses memory of region in the address range described by base, stride, and count. The actual extent of the memory addressed by the object is stride x *size of RawIntWriter* x count. The object is allocated in the current memory area of the calling thread.

region — The address space from which the new instance should be taken.

base — The starting physical address accessible through the returned instance.

count — The number of memory elements accessible through the returned instance.

stride — The distance to the next element as a mulitple of element size, where 1 means the elements are adjacent in memory.

returns an object that implements **javax.realtime.device.RawIntWriter** and supports access to the specified range in the memory region.

Throws IllegalArgumentException when base is negative, count is not greater than zero, or stride is less than one.

Throws SecurityException when the caller does not have permissions to access the given memory region or the specified range of addresses.

Throws OffsetOutOfBoundsException when base is invalid.

Throws SizeOutOfBoundsException when the memory addressed by the object would extend into an invalid range of memory.

Throws MemoryTypeConflictException when base does not point to a memory that matches the type served by this factory.

```
@SCJAllowed
@SCJMayAllocate({javax.safetycritical.annotate.AllocationContext.CURRENT})
@SCJMaySelfSuspend(false)
@SCJPhase({
    javax.safetycritical.annotate.Phase.STARTUP,
    javax.safetycritical.annotate.Phase.INITIALIZATION,
    javax.safetycritical.annotate.Phase.RUN,
    javax.safetycritical.annotate.Phase.CLEANUP })
public javax.realtime.device.RawLong createRawLong(RawMemoryRegion region,
  long base,
  int count,
  int stride)
  throws
      java.lang.SecurityException, javax.realtime.OffsetOutOfBoundsException,
      javax.realtime.SizeOutOfBoundsException,
      javax.realtime.MemoryTypeConflictException,
      javax.realtime.UnsupportedRawMemoryRegionException
```

Create an instance of a class that implements **javax.realtime.device.RawLong** and accesses memory of region in the address range described by base, stride, and count. The actual extent of the memory addressed by the object is stride x *size of RawLong* x count. The object is allocated in the current memory area of the calling thread.

region — The address space from which the new instance should be taken.

base — The starting physical address accessible through the returned instance.

count — The number of memory elements accessible through the returned instance.

stride — The distance to the next element as a mulitple of element size, where 1 means the elements are adjacent in memory.

returns an object that implements **javax.realtime.device.RawLong** and supports access to the specified range in the memory region.

Throws IllegalArgumentException when base is negative, count is not greater than zero, or stride is less than one.

Throws SecurityException when the caller does not have permissions to access the given memory region or the specified range of addresses.

Throws OffsetOutOfBoundsException when base is invalid.

Throws SizeOutOfBoundsException when the memory addressed by the object would extend into an invalid range of memory.

Throws MemoryTypeConflictException when base does not point to a memory that matches the type served by this factory.

```
@SCJAllowed
@SCJMayAllocate({javax.safetycritical.annotate.AllocationContext.CURRENT})
@SCJMaySelfSuspend(false)
@SCJPhase({
    javax.safetycritical.annotate.Phase.STARTUP,
    javax.safetycritical.annotate.Phase.INITIALIZATION,
    javax.safetycritical.annotate.Phase.RUN,
    javax.safetycritical.annotate.Phase.CLEANUP })
public javax.realtime.device.RawLongReader createRawLongReader(
  RawMemoryRegion region,
  long base,
  int count,
  int stride)
  throws
      java.lang.SecurityException, javax.realtime.OffsetOutOfBoundsException,
      javax.realtime.SizeOutOfBoundsException,
      javax.realtime.MemoryTypeConflictException,
      javax.realtime.UnsupportedRawMemoryRegionException
```

Create an instance of a class that implements **javax.realtime.device.RawLongReader** and accesses memory of region in the address range described by base, stride, and count. The actual extent of the memory addressed by the object is stride x *size of RawLongReader* x count. The object is allocated in the current memory area of the calling thread.

region — The address space from which the new instance should be taken.

base — The starting physical address accessible through the returned instance.

count — The number of memory elements accessible through the returned instance.

stride — The distance to the next element as a mulitple of element size, where 1 means the elements are adjacent in memory.

returns an object that implements **javax.realtime.device.RawLongReader** and supports access to the specified range in the memory region.

Throws IllegalArgumentException when base is negative, count is not greater than zero, or stride is less than one.

Throws SecurityException when the caller does not have permissions to access the given memory region or the specified range of addresses.

Throws OffsetOutOfBoundsException when base is invalid.

Throws SizeOutOfBoundsException when the memory addressed by the object would extend into an invalid range of memory.

Throws MemoryTypeConflictException when base does not point to a memory that matches the type served by this factory.

```
@SCJAllowed
@SCJMayAllocate({javax.safetycritical.annotate.AllocationContext.CURRENT})
@SCJMaySelfSuspend(false)
@SCJPhase({
    javax.safetycritical.annotate.Phase.STARTUP,
    javax.safetycritical.annotate.Phase.INITIALIZATION,
    javax.safetycritical.annotate.Phase.RUN,
    javax.safetycritical.annotate.Phase.CLEANUP })
public javax.realtime.device.RawLongWriter createRawLongWriter(
  RawMemoryRegion region,
  long base,
  int count,
  int stride)
  throws
      java.lang.SecurityException, javax.realtime.OffsetOutOfBoundsException,
      javax.realtime.SizeOutOfBoundsException,
      javax.realtime.MemoryTypeConflictException,
      javax.realtime.UnsupportedRawMemoryRegionException
```

Create an instance of a class that implements **javax.realtime.device.RawLongWriter** and accesses memory of region in the address range described by base, stride, and count. The actual extent of the memory addressed by the object is stride x *size of RawLongWriter* x count. The object is allocated in the current memory area of the calling thread.

region — The address space from which the new instance should be taken.

base — The starting physical address accessible through the returned instance.

count — The number of memory elements accessible through the returned instance.

stride — The distance to the next element as a mulitple of element size, where 1 means the elements are adjacent in memory.

returns an object that implements **javax.realtime.device.RawLongWriter** and supports access to the specified range in the memory region.

Throws IllegalArgumentException when base is negative, count is not greater than zero, or stride is less than one.

Throws SecurityException when the caller does not have permissions to access the given memory region or the specified range of addresses.

Throws OffsetOutOfBoundsException when base is invalid.

Throws SizeOutOfBoundsException when the memory addressed by the object would extend into an invalid range of memory.

Throws MemoryTypeConflictException when base does not point to a memory that matches the type served by this factory.

```
@SCJAllowed
@SCJMayAllocate({javax.safetycritical.annotate.AllocationContext.CURRENT})
@SCJMaySelfSuspend(false)
@SCJPhase({
    javax.safetycritical.annotate.Phase.STARTUP,
    javax.safetycritical.annotate.Phase.INITIALIZATION,
    javax.safetycritical.annotate.Phase.RUN,
    javax.safetycritical.annotate.Phase.CLEANUP })
public javax.realtime.device.RawShort createRawShort(RawMemoryRegion region,
  long base,
  int count,
  int stride)
  throws
      java.lang.SecurityException, javax.realtime.OffsetOutOfBoundsException,
      javax.realtime.SizeOutOfBoundsException,
      javax.realtime.MemoryTypeConflictException,
      javax.realtime.UnsupportedRawMemoryRegionException
```

Create an instance of a class that implements **javax.realtime.device.RawShort** and accesses memory of region in the address range described by base, stride, and count. The actual extent of the memory addressed by the object is stride x *size of RawShort* x count. The object is allocated in the current memory area of the calling thread.

region — The address space from which the new instance should be taken.

base — The starting physical address accessible through the returned instance.

count — The number of memory elements accessible through the returned instance.

stride — The distance to the next element as a mulitple of element size, where 1 means the elements are adjacent in memory.

returns an object that implements **javax.realtime.device.RawShort** and supports access to the specified range in the memory region.

Throws IllegalArgumentException when base is negative, count is not greater than zero, or stride is less than one.

Throws SecurityException when the caller does not have permissions to access the given memory region or the specified range of addresses.

Throws OffsetOutOfBoundsException when base is invalid.

Throws SizeOutOfBoundsException when the memory addressed by the object would extend into an invalid range of memory.

Throws MemoryTypeConflictException when base does not point to a memory that matches the type served by this factory.

```
@SCJAllowed
@SCJMayAllocate({javax.safetycritical.annotate.AllocationContext.CURRENT})
@SCJMaySelfSuspend(false)
@SCJPhase({
    javax.safetycritical.annotate.Phase.STARTUP,
    javax.safetycritical.annotate.Phase.INITIALIZATION,
    javax.safetycritical.annotate.Phase.RUN,
    javax.safetycritical.annotate.Phase.CLEANUP })
public javax.realtime.device.RawShortReader createRawShortReader(
  RawMemoryRegion region,
  long base,
  int count,
  int stride)
  throws
      java.lang.SecurityException, javax.realtime.OffsetOutOfBoundsException,
      javax.realtime.SizeOutOfBoundsException,
      javax.realtime.MemoryTypeConflictException,
      javax.realtime.UnsupportedRawMemoryRegionException
```

Create an instance of a class that implements **javax.realtime.device.RawShortReader** and accesses memory of region in the address range described by base, stride, and count. The actual extent of the memory addressed by the object is stride x *size of RawShortReader* x count. The object is allocated in the current memory area of the calling thread.

region — The address space from which the new instance should be taken.

base — The starting physical address accessible through the returned instance.

count — The number of memory elements accessible through the returned instance.

stride — The distance to the next element as a mulitple of element size, where 1 means the elements are adjacent in memory.

returns an object that implements **javax.realtime.device.RawShortReader** and supports access to the specified range in the memory region.

Throws IllegalArgumentException when base is negative, count is not greater than zero, or stride is less than one.

Throws SecurityException when the caller does not have permissions to access the given memory region or the specified range of addresses.

Throws OffsetOutOfBoundsException when base is invalid.

Throws SizeOutOfBoundsException when the memory addressed by the object would extend into an invalid range of memory.

Throws MemoryTypeConflictException when base does not point to a memory that matches the type served by this factory.

```
@SCJAllowed
@SCJMayAllocate({javax.safetycritical.annotate.AllocationContext.CURRENT})
@SCJMaySelfSuspend(false)
@SCJPhase({
    javax.safetycritical.annotate.Phase.STARTUP,
    javax.safetycritical.annotate.Phase.INITIALIZATION,
    javax.safetycritical.annotate.Phase.RUN,
    javax.safetycritical.annotate.Phase.CLEANUP })
public javax.realtime.device.RawShortWriter createRawShortWriter(
  RawMemoryRegion region,
  long base,
  int count,
  int stride)
  throws
      java.lang.SecurityException, javax.realtime.OffsetOutOfBoundsException,
      javax.realtime.SizeOutOfBoundsException,
      javax.realtime.MemoryTypeConflictException,
      javax.realtime.UnsupportedRawMemoryRegionException
```

Create an instance of a class that implements **javax.realtime.device.RawShortWriter** and accesses memory of region in the address range described by base, stride, and count. The actual extent of the memory addressed by the object is stride x *size of RawShortWriter* x count. The object is allocated in the current memory area of the calling thread.

region — The address space from which the new instance should be taken.

base — The starting physical address accessible through the returned instance.

count — The number of memory elements accessible through the returned instance.

stride — The distance to the next element as a mulitple of element size, where 1 means the elements are adjacent in memory.

returns an object that implements **javax.realtime.device.RawShortWriter** and supports access to the specified range in the memory region.

Throws IllegalArgumentException when base is negative, count is not greater than zero, or stride is less than one.

Throws SecurityException when the caller does not have permissions to access the given memory region or the specified range of addresses.

Throws OffsetOutOfBoundsException when base is invalid.

Throws SizeOutOfBoundsException when the memory addressed by the object would extend into an invalid range of memory.

Throws MemoryTypeConflictException when base does not point to a memory that matches the type served by this factory.

```
@SCJAllowed
@SCJMayAllocate({})
@SCJMaySelfSuspend(false)
@SCJPhase({
    javax.safetycritical.annotate.Phase.STARTUP,
    javax.safetycritical.annotate.Phase.INITIALIZATION,
    javax.safetycritical.annotate.Phase.RUN,
    javax.safetycritical.annotate.Phase.CLEANUP })
public void deregister(RawMemoryRegionFactory factory)
  throws javax.realtime.DeregistrationException
```

Remove support for a new memory region

factory — is the **javax.realtime.device.RawMemoryRegionFactory** to make unavailable.

Throws RegistrationException when the factory is not registered.

```
@SCJAllowed
@SCJMaySelfSuspend(false)
@SCJPhase({
    javax.safetycritical.annotate.Phase.STARTUP,
    javax.safetycritical.annotate.Phase.INITIALIZATION,
    javax.safetycritical.annotate.Phase.RUN,
    javax.safetycritical.annotate.Phase.CLEANUP})
@SCJMayAllocate({})
public static javax.realtime.device.RawMemoryFactory getDefaultFactory( )
```

Get the factory with support for the platform defined regions.

returns the platform defined factory

```
@SCJAllowed
@SCJMayAllocate({})
@SCJMaySelfSuspend(false)
@SCJPhase({
    javax.safetycritical.annotate.Phase.STARTUP,
    javax.safetycritical.annotate.Phase.INITIALIZATION,
    javax.safetycritical.annotate.Phase.RUN,
    javax.safetycritical.annotate.Phase.CLEANUP })
public void register(RawMemoryRegionFactory factory)
  throws javax.realtime.RegistrationException
```

Add support for a new memory region

factory — is the **javax.realtime.device.RawMemoryRegionFactory** to use for creating **javax.realtime.device.RawMemory** objects for the **javax.realtime.device-.RawMemoryRegion** s it makes available.

Throws RegistrationException when the factory already is already registered.

5.2.19 javax.realtime.device.InterruptServiceRoutine

This class is a restricted version of the class provided by the RTSJ specification.

Declaration

```
@SCJAllowed(javax.safetycritical.annotate.Level.LEVEL_1)
 public abstract class InterruptServiceRoutine implements
     javax.realtime.BoundRealtimeExecutor extends java.lang.Object
```

Description

A first level interrupt handling mechanisms. Override the handle method to provide the first level interrupt handler. The constructors for this class are invoked by the infrastructure and are therefore not visible to the application. The default affinity of an handler can be determined via calling **getAffinity()** .

Methods

```
@SCJAllowed(javax.safetycritical.annotate.Level.LEVEL_1)
@SCJPhase({
   javax.safetycritical.annotate.Phase.INITIALIZATION,
   javax.safetycritical.annotate.Phase.RUN,
   javax.safetycritical.annotate.Phase.CLEANUP})
@SCJMaySelfSuspend(false)
@SCJMayAllocate({
   javax.safetycritical.annotate.AllocationContext.CURRENT,
   javax.safetycritical.annotate.AllocationContext.INNER,
   javax.safetycritical.annotate.AllocationContext.OUTER})
public static javax.realtime.device.InterruptServiceRoutine getHandler(
  int interrupt)
```

Find the InterruptServiceRoutine that is handling a given interrupt.

interrupt — for which to find the InterruptServiceRoutine

returns the InterruptServiceRoutine registered to the given interrupt. Null is returned when nothing is registered for that interrupt.

```
@SCJAllowed(javax.safetycritical.annotate.Level.LEVEL_1)
@SCJMayAllocate({})
@SCJMaySelfSuspend(false)
@SCJPhase({
   javax.safetycritical.annotate.Phase.STARTUP,
   javax.safetycritical.annotate.Phase.INITIALIZATION,
   javax.safetycritical.annotate.Phase.RUN,
   javax.safetycritical.annotate.Phase.CLEANUP })
public static int getInterruptPriority(int InterruptId)
```

Every interrupt has an implementation-defined integer id.

returns The priority of the code that the first-level interrupts code executes. The returned value is always greater than PriorityScheduler.getMaxPriority().

Throws IllegalArgumentException if unsupported InterruptId

```
@SCJAllowed(javax.safetycritical.annotate.Level.LEVEL_1)
@SCJPhase({
    javax.safetycritical.annotate.Phase.INITIALIZATION,
    javax.safetycritical.annotate.Phase.RUN,
    javax.safetycritical.annotate.Phase.CLEANUP})
@SCJMaySelfSuspend(false)
@SCJMayAllocate({
    javax.safetycritical.annotate.AllocationContext.CURRENT,
    javax.safetycritical.annotate.AllocationContext.INNER,
    javax.safetycritical.annotate.AllocationContext.OUTER})
public static int getMaximumInterruptPriority( )
```

Retrieve the maximum interrupt priority. It must be greater than or equal to the result of getMinimumInterruptPriority.

returns the maximum interrupt priority.

```
@SCJAllowed(javax.safetycritical.annotate.Level.LEVEL_1)
@SCJPhase({
    javax.safetycritical.annotate.Phase.INITIALIZATION,
    javax.safetycritical.annotate.Phase.RUN,
    javax.safetycritical.annotate.Phase.CLEANUP})
@SCJMaySelfSuspend(false)
@SCJMayAllocate({
    javax.safetycritical.annotate.AllocationContext.CURRENT,
    javax.safetycritical.annotate.AllocationContext.INNER,
    javax.safetycritical.annotate.AllocationContext.OUTER})
public static int getMinimumInterruptPriority( )
```

Retrieve the minimum interrupt priority. It must be higher than all other priorities provided by the system.

returns the minimum interrupt priority.

```
@SCJAllowed(javax.safetycritical.annotate.Level.LEVEL_1)
@SCJPhase({
    javax.safetycritical.annotate.Phase.INITIALIZATION,
    javax.safetycritical.annotate.Phase.RUN,
    javax.safetycritical.annotate.Phase.CLEANUP})
@SCJMaySelfSuspend(false)
@SCJMayAllocate({javax.safetycritical.annotate.AllocationContext.CURRENT})
protected abstract void handle( )
```

The code to execute for first level interrupt handling. A subclass defines this to give the proper behavior. No code that could self-suspend may be called here. The effects of unbound blocking and inducing a context switch here are undefined and could result in deadlocking the machine. Unless the overridden method is synchronized, the infrastructure shall provide no synchronization for the execution of this method.

5.2.20 javax.safetycritical.ManagedInterruptServiceRoutine

This class integrates the RTSJ interrupt handling mechanisms with the SCJ mission structure.

Declaration

```
@SCJAllowed(javax.safetycritical.annotate.Level.LEVEL_1)
 public abstract class ManagedInterruptServiceRoutine extends
     javax.realtime.device.InterruptServiceRoutine
```

Constructors

```
@SCJAllowed(javax.safetycritical.annotate.Level.LEVEL_1)
@SCJMaySelfSuspend(false)
@SCJMayAllocate({})
@SCJPhase({javax.safetycritical.annotate.Phase.INITIALIZATION})
public ManagedInterruptServiceRoutine(long sizes)
```

Creates an interrupt service routine

sizes — defines the memory space required by the handle method.

Methods

```
@SCJAllowed(javax.safetycritical.annotate.Level.LEVEL_1)
@SCJPhase({
   javax.safetycritical.annotate.Phase.INITIALIZATION,
   javax.safetycritical.annotate.Phase.RUN,
   javax.safetycritical.annotate.Phase.CLEANUP})
@SCJMaySelfSuspend(false)
@SCJMayAllocate({})
@Override
public javax.realtime.Affinity getAffinity( )
```

Determine the affinity set instance associated with {@code task}.

returns The associated affinity.

```
@SCJAllowed(javax.safetycritical.annotate.Level.LEVEL_1)
@Override
@SCJPhase({javax.safetycritical.annotate.Phase.INITIALIZATION})
@SCJMaySelfSuspend(false)
@SCJMayAllocate({})
public final void register(int interrupt)
  throws javax.realtime.RegistrationException
```

Equivalent to register(interrupt, prio) where prio is the highest InterruptCeilingPriority defined.

Throws IllegalStateException if this method is not part of a Mission which is currently being initialized

```
@SCJAllowed(javax.safetycritical.annotate.Level.LEVEL_1)
@SCJPhase({javax.safetycritical.annotate.Phase.INITIALIZATION})
@SCJMaySelfSuspend(false)
@SCJMayAllocate({})
public final void register(int interrupt, int ceiling)
  throws javax.realtime.RegistrationException
```

Registers the ISR for the given interrupt with the current mission and sets the ceiling priority of this. The filling of the associated interrupt vector is deferred until the end of the initialisation phase.

interrupt — is the implementation-dependent id for the interrupt.

ceiling — is the required ceiling priority.

Throws IllegalArgumentException if the ceiling is lower than the interrupt priority.

Throws RegistrationException if this object is not part of a Mission which is currently being initialized.

```
@SCJAllowed(javax.safetycritical.annotate.Level.LEVEL_1)
@SCJPhase({javax.safetycritical.annotate.Phase.INITIALIZATION})
@SCJMaySelfSuspend(false)
@SCJMayAllocate({})
@Override
public void setAffinity(Affinity set)
  throws java.lang.IllegalArgumentException,
    javax.realtime.ProcessorAffinityException, java.lang.NullPointerException
```

Set the processor affinity of a {@code task} to {@code set} with immediate effect.

set — is the processor affinity

Throws IllegalArgumentException when the intersection of {@code set} the affinity of any {@code ThreadGroup} instance containing {@code task} is empty.

Throws ProcessorAffinityException is thrown when the runtime fails to set the affinity for platform-specific reasons.

Throws NullPointerException when {@code set} is {@code null}.

```
@SCJAllowed(javax.safetycritical.annotate.Level.LEVEL_1)
@SCJPhase({javax.safetycritical.annotate.Phase.RUN})
@SCJMaySelfSuspend(false)
@SCJMayAllocate({})
public void unhandledException(Exception except)
```

Called by the infrastructure if an exception propagates outside of the handle method.

except — is the uncaught exception.

```
@SCJAllowed(javax.safetycritical.annotate.Level.LEVEL_1)
@SCJAllowed(javax.safetycritical.annotate.Level.SUPPORT)
@SCJPhase({javax.safetycritical.annotate.Phase.CLEANUP})
@SCJMaySelfSuspend(false)
@SCJMayAllocate({})
@Override
public final void unregister( )
```

Unregisters the ISR with the current mission.

5.3 POSIX Signal Handlers

In the RTSJ, all asynchronous external events are associated with *active events*. The simpler SCJ specification does not support the full generality of the RTSJ active event model. Instead, SCJ provides a specialized API that maps POSIX signals to asynchronous event handlers.

5.3.1 Semantics and Requirements

POSIX signals, as defined in POSIX IEEE Std 1003.1-2008, provide a mechanism to asynchronously notify particular processes or specific threads within particular processes of events that deserve their attention. The POSIX standard defines the names of standard signals, and allows each POSIX implementation to define a mapping between the signal name and an integer signal identification number. Application

programs use the signal number to register handlers that execute whenever notification of the associated signal is delivered to the application process. Since the POSIX standard allows multiple signal names to map to the same signal number, it is not always possible for applications to differentiate between the events that release a particular signal handler.

The POSIX standard does not require every POSIX implementation to support all of the standard signals. Furthermore, the standard allows platforms to support non-standard signals, known by platform-specific names. As part of a POSIX platform's configuration, the platform implementer must define the names and numeric identities of all of the signals supported by that platform.

- An implementation shall define all the POSIX signals and real-time signals that it supports.
- The SCJ specification does not require an implementation to allow external POSIX processes to deliver POSIX signals to specific ManagedSchedulables.
- Each signal has an associated integer id and a string name. These values are defined by the underlying platform, in accordance with the POSIX standard.
- For each signal, the default behavior of the handler is as specified by POSIX.
- For each signal, the application shall be able to set a single managed handler.
- Only one handler may be associated with each signal id.
- For POSIX signals, the parameter passed to the handleAsyncEvent shall be the id of the signal that is being handled.
- For POSIX real-time signals, the parameter passed to the handleAsyncEvent shall be the payload (siginfo_t) associated with the signal that was generated.

5.3.2 Level Considerations

Level 0

Signal handlers of any kind are prohibited at Level 0.

Level 1

Signal handlers shall be supported.

5.3.3 javax.safetycritical.POSIXSignalHandler

Declaration

```
@SCJAllowed(javax.safetycritical.annotate.Level.LEVEL_1)
 public abstract class POSIXSignalHandler extends
     javax.safetycritical.ManagedEventHandler
```

Description

This class enables the automatic execution of code that is bound to a real-time POSIX signal. It is abstract. Concrete subclasses must implement the handleAsyncEvent method and may override the default cleanUp method. The parameter passed by the infrastructure to the handleAsyncEvent method is the id of the caught signal.

Constructors

```
@SCJAllowed(javax.safetycritical.annotate.Level.LEVEL_1)
@SCJPhase({javax.safetycritical.annotate.Phase.INITIALIZATION})
@SCJMaySelfSuspend(false)
@SCJMayAllocate({javax.safetycritical.annotate.AllocationContext.CURRENT})
public POSIXSignalHandler(PriorityParameters priority,
  AperiodicParameters release,
  ScopeParameters storage,
  ConfigurationParameters config,
  int signalId)
```

Constructs a real-time POSIX signal handler that will be released when the signal is delivered.

The values passed as constructor parameters are captured at construction time. Any subsequent mutation of the parameter objects has no effect on the behavior of the constructed object.

priority — specifies the priority parameters for this handler; it must not be null.

release — specifies the release parameters for this handler. A null parameter indicates that there is no deadline associated with this handler.

storage — specifies the ScopeParameters for this handler; it must not be null

config — specifies the ConfigurationParameters for this handler

signalId — specifies the signal that releases this handler.

Throws IllegalArgumentException when priority or storage is null; or when signalIds already has an attached handler or the signalId is outside the range of POSIX signals.

Methods

```
@SCJAllowed(javax.safetycritical.annotate.Level.LEVEL_1)
@SCJMaySelfSuspend(false)
@SCJMayAllocate({})
```

```
@SCJPhase({
    javax.safetycritical.annotate.Phase.STARTUP,
    javax.safetycritical.annotate.Phase.INITIALIZATION,
    javax.safetycritical.annotate.Phase.RUN,
    javax.safetycritical.annotate.Phase.CLEANUP })
public static int getSignalId(String name)
```

Get the POSIX signal id represented by name.

name — The name of the POSIX signal.

returns The id of the POSIX signal whose name is name

Throws IllegalArgumentException if there is no POSIX signal with this name.

5.3.4 javax.safetycritical.POSIXRealtimeSignalHandler

Declaration

```
@SCJAllowed(javax.safetycritical.annotate.Level.LEVEL_1)
 public abstract class POSIXRealtimeSignalHandler extends
     javax.safetycritical.ManagedLongEventHandler
```

Description

This class permits the automatic execution of code that is bound to a real-time POSIX signal. It is abstract. Concrete subclasses must implement the handleAsyncEvent method and may override the default cleanUp method. The parameter passed by the infrastructure to the handleAsyncEvent method is the id of the caught signal.

Constructors

```
@SCJAllowed(javax.safetycritical.annotate.Level.LEVEL_1)
@SCJPhase({javax.safetycritical.annotate.Phase.INITIALIZATION})
@SCJMaySelfSuspend(false)
@SCJMayAllocate({javax.safetycritical.annotate.AllocationContext.CURRENT})
public POSIXRealtimeSignalHandler(PriorityParameters priority,
  AperiodicParameters release,
  ScopeParameters storage,
  ConfigurationParameters config,
  int signalId)
```

Constructs an real-time POSIX signalt handler that will be released when the signal is delivered.

The values passed as constructor parameters are captured at construction time. Any subsequent mutation of the parameter objects has no effect on the behavior of the constructed object.

priority — specifies the priority parameters for this handle; it must not be null.

release — specifies the release parameters for this handler. A null parameter indicates that there is no deadline associated with this handler.

storage — specifies the ScopeParameters for this handler; it must not be null

config — specifies the ConfigurationParameters for this handler

signalId — specifies the id of the POSIX real-time signal that releases this handler.

Throws IllegalArgumentException when priority or storage is null; or when the signalId already has an attached handler or the signalId is outside the range of POSIX real-time signals.

Methods

```
@SCJAllowed(javax.safetycritical.annotate.Level.LEVEL_1)
@SCJMaySelfSuspend(false)
@SCJMayAllocate({})
@SCJPhase({
    javax.safetycritical.annotate.Phase.STARTUP,
    javax.safetycritical.annotate.Phase.INITIALIZATION,
    javax.safetycritical.annotate.Phase.RUN,
    javax.safetycritical.annotate.Phase.CLEANUP })
public static int getSignalId(String name)
```

Get the POSIX real-time signal id represented by name.

name — The name of the POSIX real-time signal.

returns The id of the POSIX real-time signal with this name

Throws IllegalArgumentException if there is no POSIX real-time signal with this name.

5.4 Rationale

Many safety-critical real-time systems must interact with the embedded environment. This can be done either at a low level through device registers and interrupt handling, or via some higher-level input and output mechanisms.

There are at least four execution (run-time) environments for SCJ:

1. On top of a high-integrity real-time operating system where the Java application runs in user mode.

2. As part of an embedded device where the Java application runs stand-alone on a hardware/software virtual machine.

3. As a "kernel module" incorporated into a high-integrity real-time kernel where both kernel and application run in supervisor mode.

4. As a stand-alone cyclic executive with minimal operating system support.

In execution environment (1), interaction with the embedded environment will usually be via operating system calls using connection-oriented APIs. The Java program will typically have no direct access to the IO devices (although some limited access to physical memory may be provided, it is unlikely that interrupts can be directly handled). Connection-oriented input output mechanisms are discussed in Chapter 6.

In execution environments (2), (3) and (4), the Java program may be able to directly access devices and handle interrupts. Such low-level device access is the topic of this chapter.

A device can be considered to be a processor performing a fixed task. Therefore, a computer system can be considered to be a collection of parallel threads. There are several models by which the device 'thread' can communicate and synchronize with the tasks executing inside the main processor. All models must provide[1]:

1. **A suitable representation of interrupts** (if interrupts are to be handled), and

2. **Facilities for representing, addressing and manipulating device registers**.

In the RTSJ and SCJ, the former is provided by interrupt service routines The RTSJ physical and raw memory access facilities allow broad support for accessing memory with different characteristics. SCJ restricts these facilities to focus on those that can be used for accessing registers that are both memory mapped and port mapped.

5.4.1 Stride

Since the word size of devices do not always match the word size of the memory or I/O bus, the interface provides for the notion of stride. Stride defines the distance between elements in a raw memory area. Normally elements of a memory area are mapped sequentially, without any space between the elements. This is a stride of one. A stride of two, means that every other element in physical memory is mapped into the raw memory area.

For example, it is often easier to map a 16 bit device into a 32 bit system by mapping the 16 bit registers at 32 bit intervals. This enables 16 bit accesses to the device to be atomic on 32 bit addressed systems, even when the bus always does 32 bit transfers. One can create a RawShort area with a stride of two. Then the area can be accessed as if the registers where contiguous.

5.4.2 Interrupt Handling Rationale

The SCJ Interrupt Handling model is heavily influenced by the Ada interrupt handling model, and borrows most of its semantics from that model. Interrupt handling is necessarily machine dependent. However, SCJ tries to provide an abstract model that can be implemented on top of all architectures. The model assumes that:

- The processor has a (logical) interrupt controller chip that monitors a number of *interrupt lines*;
- Each interrupt line has an associated interrupt priority;
- Associated with the interrupt lines is a (logical) interrupt vector that contains the addresses of the interrupt service routines;
- The processor has instructions that allow interrupts from a particular line to be disabled/masked irrespective of the type of device attached;
- Disabling interrupts from a specific line may, or may not, disable the interrupts from lines of lower priority;
- A device can be connected to an arbitrary interrupt line;
- When an interrupt is signalled on an interrupt line by a device, the handling processor uses the identity of the interrupt line to index into the interrupt vector and jump to the address of the interrupt service routine. The processor automatically disables further interrupts (either of the same priority or, possibly, all interrupts) on that processor).
- On return from the interrupt service routine, interrupts are automatically re-enabled.

For each of the interrupt priorities, SCJ has an associated hardware priority that can be used to set the ceiling of an ISR object. The SCJ infrastructure uses this to disable

the interrupts from the associated interrupt line, and lower priority interrupts, when it is executing a synchronized method of the object. For the handle method, this may be done automatically by the hardware interrupt handling mechanism or it may require added support from the infrastructure. However, for clarity of the model, SCJ recommends that the handle method should be defined as synchronized. Similarly, although the unhandledException method, if called, will be called with interrupts disabled, for clarity it should be defined as synchronized as well. The SCJ allows an SCJ byte code verifier to flag an error if these methods are not synchronized.

SCJ indicates that the application should refrain from memory allocations in an outer-nested immortal or mission memory area while it is executing in an ISR synchronized method. This is because such allocations are likely to require a lock associated with these memory areas. The time taken to acquire that lock may be significant compared to any latency requirement on interrupt handling. The infrastructure does not guarantee the ceilings of the shared memory regions is in the interrupt priority range, hence ceiling violation may occur.

5.5 Compatibility

The SCJ interrupt handling facility uses the same model as the RTSJ.

Chapter 6

Input and Output Model

6.1 Overview

Safety-critical systems often have limited input and output capabilities. This makes it difficult to provide a common set of I/O classes for safety-critical applications. The standard file and socket classes are too heavy weight for many safety-critical systems. However, the Java Micro Edition provides a basis for a flexible I/O mechanism that is much leaner than that of other Java configurations, so SCJ. uses a subset of it as a simple, extendable I/O capability.

The Overview and Rationale sections are not normative but are provided to improve understanding of the normative sections. All of the other sections of this chapter are normative.

6.2 Semantics and Requirements

Since there is no common I/O facility that can be found on every safety-critical system, a flexible mechanism for I/O capabilities is needed. The Java Micro Edition I/O Connector and Connection classes, with the StreamConnection, InputConnection, and OutputConnection interfaces, provide a good basis. Figure 6.1 gives an overview of the I/O interfaces and classes provided by SCJ.

The Java Micro Edition's Connector class does not directly support extensibility. Therefore, SCJ provides an additional class, ConnectionFactory to provide the framework for application-defined connection types, which can be registered with ConnectionFactory and instantiated by the standard Connector class.

A Connector maps a URL string to a factory for creating a Connection for the given URL. The protocol part of a URL passed to Connector, e.g. http at the beginning of

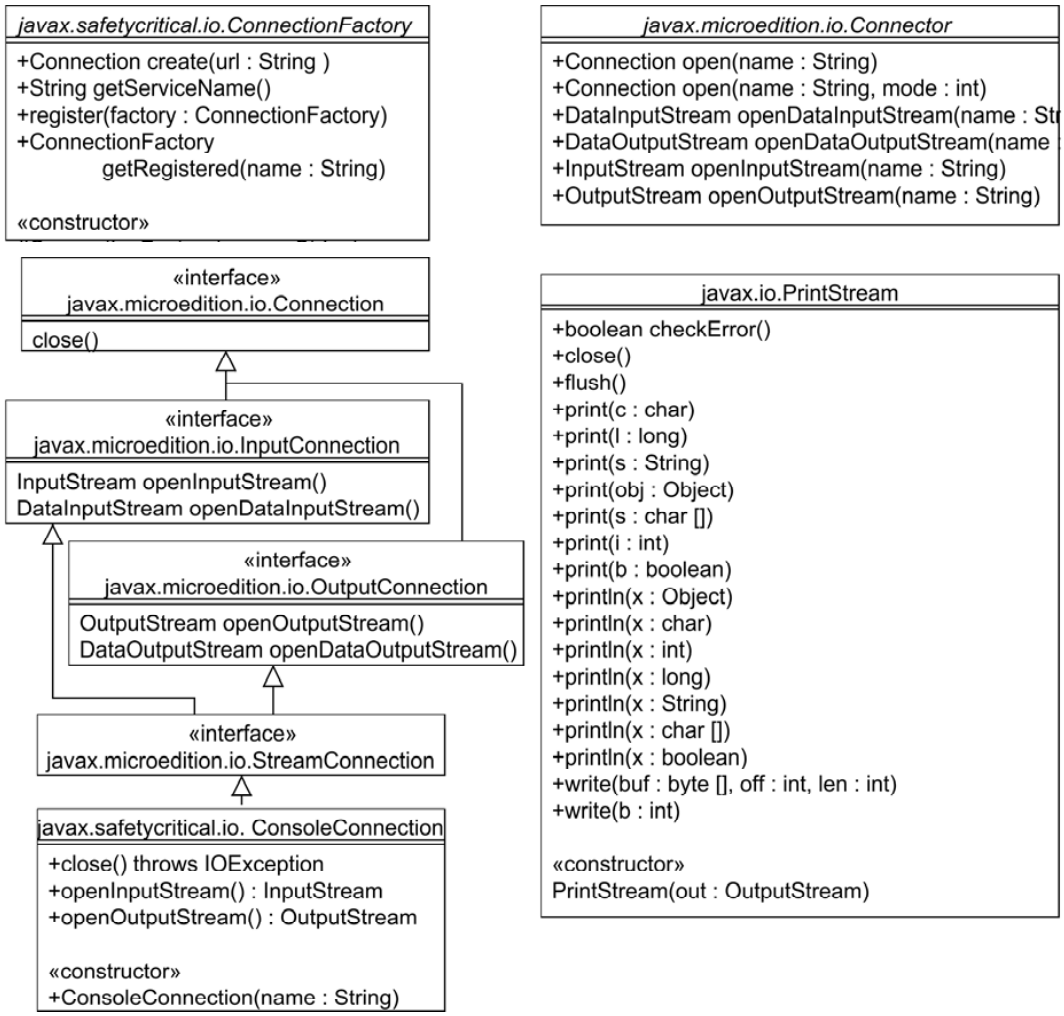

Figure 6.1: Interfaces and classes supporting streaming I/O

a web address, is used to select the proper factory. The rest of the URL is used as arguments to the factory to create a connection of the proper type.

Within SCJ, the protocol console defines the default console. The console can be used to read from and send output to some implementation-defined data source or sink external to the SCJ implementation. The console connection is represented by the ConsoleConnection class.

An SCJ implementation shall support the console connection. In the simplest case the console connection represents a serial line interface, but can also represent a buffer in memory. The test harnesses within the SCJ Technology Compatibility Kit (TCK) use console for the test output.

In addition to ConsoleConnection, a simplified version of java.io.PrintStream is pro-

vided by SCJ. With these classes, every safety-critical system has an I/O facility, even if it is just to and from a memory buffer.

6.3 Level Considerations

The I/O classes are available for all SCJ compliance levels.

6.4 API

SCJ supports the connection framework of of the Java Micro Edition as defined in package javax.microedition.io. Additional classes are provided in javax.safetycritical.io for a console connection, a simple printer filter, and a factory to create user defined connection types.

Unless indicated otherwise, the classes defined in this section are thread safe

6.4.1 javax.microedition.io.Connector

Declaration

```
@SCJAllowed
 public class Connector extends java.lang.Object
```

Description

This class is a factory for use by applications to dynamically create Connection objects. The application provides a specified name that this factory will use to identify an appropriate connection to a device or interface. The specified name conforms to the URL format defined in RFC 2396. The specified name uses this format:

{scheme}:[{target}][{params}]

where {scheme} is the name of a protocol such as *http* .

The {target} is normally some kind of network address or other interface such as a file designation.

Any {params} are formed as a series of equates of the form ";x=y". Example: ";type=a".

Within this format, the application may provide an optional second parameter to the open function. This second parameter is a mode flag to indicate the intentions of the calling code to the protocol handler. The options here specify

whether the connection will be used to read (READ), write (WRITE), or both (READ_WRITE). Each protocol specifies which flag settings are permitted. For example, a printer would likely not permit read access, so it might throw an IllegalArgumentException. If not specified, READ_WRITE mode is used by default. // *

// * In addition, a third parameter may be specified as a boolean flag // * indicating that the application intends to handle timeout exceptions. // * If this flag is true, the protocol implementation may throw an // * InterruptedIOException if a timeout condition is detected. // * This flag may be ignored by the protocol handler; the // * InterruptedIOException may not actually be thrown. // * If this parameter is false, the protocol shall not throw // * the InterruptedIOException.

Fields

@SCJAllowed
public static final int READ

Access mode READ.

@SCJAllowed
public static final int READ_WRITE

Access mode READ_WRITE.

@SCJAllowed
public static final int WRITE

Access mode WRITE.

Methods

```
@SCJAllowed
@SCJMaySelfSuspend(true)
@SCJMayAllocate({
    javax.safetycritical.annotate.AllocationContext.CURRENT,
    javax.safetycritical.annotate.AllocationContext.INNER,
    javax.safetycritical.annotate.AllocationContext.OUTER})
@SCJPhase({
    javax.safetycritical.annotate.Phase.STARTUP,
    javax.safetycritical.annotate.Phase.INITIALIZATION,
    javax.safetycritical.annotate.Phase.RUN,
    javax.safetycritical.annotate.Phase.CLEANUP })
public static javax.microedition.io.Connection open(String name)
  throws java.io.IOException
      , java.lang.SecurityException
```

Create and open a Connection. This method is the same as calling open(name, READ_WRITE).

name — The URL for the connection.

returns a new Connection object.

Throws IllegalArgumentException if a parameter is invalid.

Throws ConnectionNotFoundException if the target of the name cannot be found, or if the requested protocol type is not supported.

Throws IOException if some other kind of I/O error occurs.

Throws SecurityException if access to the protocol handler is prohibited.

```
@SCJAllowed
@SCJMaySelfSuspend(true)
@SCJMayAllocate({
    javax.safetycritical.annotate.AllocationContext.CURRENT,
    javax.safetycritical.annotate.AllocationContext.INNER,
    javax.safetycritical.annotate.AllocationContext.OUTER})
@SCJPhase({
    javax.safetycritical.annotate.Phase.STARTUP,
    javax.safetycritical.annotate.Phase.INITIALIZATION,
    javax.safetycritical.annotate.Phase.RUN,
    javax.safetycritical.annotate.Phase.CLEANUP })
public static javax.microedition.io.Connection open(String name, int mode)
  throws java.io.IOException
      , java.lang.SecurityException
```

Create and open a Connection with a specified name and access mode.

name — The URL for the connection.

mode — The access mode (i.e., READ, WRITE, or READ_WRITE.)

returns A new Connection object.

Throws IllegalArgumentException if a parameter is invalid.

Throws ConnectionNotFoundException if the target of the name cannot be found, or if the requested protocol type is not supported.

Throws IOException if some other kind of I/O error occurs.

Throws SecurityException if access to the protocol handler is prohibited.

```
@SCJAllowed
@SCJMaySelfSuspend(true)
@SCJMayAllocate({
    javax.safetycritical.annotate.AllocationContext.CURRENT,
```

```
    javax.safetycritical.annotate.AllocationContext.INNER,
    javax.safetycritical.annotate.AllocationContext.OUTER})
@SCJPhase({
    javax.safetycritical.annotate.Phase.STARTUP,
    javax.safetycritical.annotate.Phase.INITIALIZATION,
    javax.safetycritical.annotate.Phase.RUN,
    javax.safetycritical.annotate.Phase.CLEANUP })
public static java.io.DataInputStream openDataInputStream(String name)
  throws
     java.io.IOException, java.lang.SecurityException
```

Create and open a connection input stream.

name — The URL for the connection.

returns A DataInputStream.

Throws IllegalArgumentException if a parameter is invalid.

Throws ConnectionNotFoundException if the target of the name cannot be found, or if the requested protocol type is not supported.

Throws IOException if some other kind of I/O error occurs.

Throws SecurityException if access to the protocol handler is prohibited.

```
@SCJAllowed
@SCJMaySelfSuspend(true)
@SCJMayAllocate({
    javax.safetycritical.annotate.AllocationContext.CURRENT,
    javax.safetycritical.annotate.AllocationContext.INNER,
    javax.safetycritical.annotate.AllocationContext.OUTER})
@SCJPhase({
    javax.safetycritical.annotate.Phase.STARTUP,
    javax.safetycritical.annotate.Phase.INITIALIZATION,
    javax.safetycritical.annotate.Phase.RUN,
    javax.safetycritical.annotate.Phase.CLEANUP })
public static java.io.DataOutputStream openDataOutputStream(String name)
  throws
     java.io.IOException, java.lang.SecurityException
```

Create and open a connection output stream.

name — The URL for the connection.

returns A DataOutputStream.

Throws IllegalArgumentException if a parameter is invalid.

Throws ConnectionNotFoundException if the target of the name cannot be found, or if the requested protocol type is not supported.

Throws IOException if some other kind of I/O error occurs.

Throws SecurityException if access to the protocol handler is prohibited.

```
@SCJAllowed
@SCJMaySelfSuspend(true)
@SCJMayAllocate({
    javax.safetycritical.annotate.AllocationContext.CURRENT,
    javax.safetycritical.annotate.AllocationContext.INNER,
    javax.safetycritical.annotate.AllocationContext.OUTER})
@SCJPhase({
    javax.safetycritical.annotate.Phase.STARTUP,
    javax.safetycritical.annotate.Phase.INITIALIZATION,
    javax.safetycritical.annotate.Phase.RUN,
    javax.safetycritical.annotate.Phase.CLEANUP })
public static java.io.InputStream openInputStream(String name)
  throws java.io.IOException,
      java.lang.SecurityException
```

Create and open a connection input stream.

name — The URL for the connection.

returns An InputStream.

Throws IllegalArgumentException if a parameter is invalid.

Throws ConnectionNotFoundException if the target of the name cannot be found, or if the requested protocol type is not supported.

Throws IOException if some other kind of I/O error occurs.

Throws SecurityException if access to the protocol handler is prohibited.

```
@SCJAllowed
@SCJMaySelfSuspend(true)
@SCJMayAllocate({
    javax.safetycritical.annotate.AllocationContext.CURRENT,
    javax.safetycritical.annotate.AllocationContext.INNER,
    javax.safetycritical.annotate.AllocationContext.OUTER})
@SCJPhase({
    javax.safetycritical.annotate.Phase.STARTUP,
    javax.safetycritical.annotate.Phase.INITIALIZATION,
    javax.safetycritical.annotate.Phase.RUN,
    javax.safetycritical.annotate.Phase.CLEANUP })
public static java.io.OutputStream openOutputStream(String name)
  throws java.io.IOException
      , java.lang.SecurityException
```

Create and open a connection output stream.

name — The URL for the connection.

returns An OutputStream.

Throws IllegalArgumentException if a parameter is invalid.

Throws ConnectionNotFoundException if the target of the name cannot be found, or if the requested protocol type is not supported.

Throws IOException if some other kind of I/O error occurs.

Throws SecurityException if access to the protocol handler is prohibited.

6.4.2 javax.microedition.io.Connection

Declaration

```
@SCJAllowed
 public interface Connection
```

Description

This is the most basic type of generic connection. Only the close method is defined. No open method is defined here because opening is always done using the Connector.open() methods.

Methods

```
@SCJAllowed
@SCJMaySelfSuspend(true)
@SCJMayAllocate({
   javax.safetycritical.annotate.AllocationContext.CURRENT,
   javax.safetycritical.annotate.AllocationContext.INNER,
   javax.safetycritical.annotate.AllocationContext.OUTER})
@SCJPhase({
   javax.safetycritical.annotate.Phase.STARTUP,
   javax.safetycritical.annotate.Phase.INITIALIZATION,
   javax.safetycritical.annotate.Phase.RUN,
   javax.safetycritical.annotate.Phase.CLEANUP })
public void close( )
```

Close the connection.

When a connection has been closed, access to any of its methods that involve an I/O operation will cause an IOException to be thrown. Closing an already closed connection has no effect. Streams derived from the connection may be open when the method is called. Any open streams will cause the connection to be held open until they themselves are closed. In this latter case access to the open streams is permitted, but access to the connection is not.

Throws IOException if an I/O error occurs

Throws IllegalArgumentException

Throws ConnectionNotFoundException

6.4.3 javax.microedition.io.InputConnection

Declaration

```
@SCJAllowed
 public interface InputConnection extends javax.microedition.io.Connection
```

Description

This interface defines the capabilities that an input stream connection must have.

Methods

```
@SCJAllowed
@SCJMayAllocate({
    javax.safetycritical.annotate.AllocationContext.CURRENT,
    javax.safetycritical.annotate.AllocationContext.INNER,
    javax.safetycritical.annotate.AllocationContext.OUTER})
@SCJMaySelfSuspend(true)
@SCJPhase({
    javax.safetycritical.annotate.Phase.STARTUP,
    javax.safetycritical.annotate.Phase.INITIALIZATION,
    javax.safetycritical.annotate.Phase.RUN,
    javax.safetycritical.annotate.Phase.CLEANUP })
public java.io.DataInputStream openDataInputStream( )
```

Open and return a data input stream for a connection.

returns An input stream.

Throws IOException if an I/O error occurs.

```
@SCJAllowed
@SCJMaySelfSuspend(true)
@SCJMayAllocate({
    javax.safetycritical.annotate.AllocationContext.CURRENT,
    javax.safetycritical.annotate.AllocationContext.INNER,
    javax.safetycritical.annotate.AllocationContext.OUTER})
@SCJPhase({
    javax.safetycritical.annotate.Phase.STARTUP,
    javax.safetycritical.annotate.Phase.INITIALIZATION,
    javax.safetycritical.annotate.Phase.RUN,
    javax.safetycritical.annotate.Phase.CLEANUP })
public java.io.InputStream openInputStream( )
```

Open and return an input stream for a connection.

returns An input stream.

Throws IOException if an I/O error occurs.

6.4.4 javax.microedition.io.OutputConnection

Declaration

```
@SCJAllowed
 public interface OutputConnection extends javax.microedition.io.Connection
```

Description

This interface defines the capabilities that an output stream connection must have.

Methods

```
@SCJAllowed
@SCJMayAllocate({
    javax.safetycritical.annotate.AllocationContext.CURRENT,
    javax.safetycritical.annotate.AllocationContext.INNER,
    javax.safetycritical.annotate.AllocationContext.OUTER})
@SCJMaySelfSuspend(true)
@SCJPhase({
    javax.safetycritical.annotate.Phase.STARTUP,
    javax.safetycritical.annotate.Phase.INITIALIZATION,
    javax.safetycritical.annotate.Phase.RUN,
    javax.safetycritical.annotate.Phase.CLEANUP })
public java.io.DataOutputStream openDataOutputStream( )
```

Open and return a data output stream for a connection.

returns An output stream.

Throws IOException if an I/O error occurs.

```
@SCJAllowed
@SCJMaySelfSuspend(true)
@SCJMayAllocate({
    javax.safetycritical.annotate.AllocationContext.CURRENT,
    javax.safetycritical.annotate.AllocationContext.INNER,
    javax.safetycritical.annotate.AllocationContext.OUTER})
@SCJPhase({
    javax.safetycritical.annotate.Phase.STARTUP,
    javax.safetycritical.annotate.Phase.INITIALIZATION,
    javax.safetycritical.annotate.Phase.RUN,
    javax.safetycritical.annotate.Phase.CLEANUP })
public java.io.OutputStream openOutputStream( )
```

Open and return an output stream for a connection.

returns An output stream.

Throws IOException if an I/O error occurs.

6.4.5 javax.microedition.io.StreamConnection

Declaration

```
@SCJAllowed
 public interface StreamConnection extends javax.microedition.io.InputConnection,
     javax.microedition.io.OutputConnection
```

Description

This interface defines the capabilities that a stream connection must have.

In a typical implementation of this interface, all StreamConnections have one underlying InputStream and one OutputStream. Opening a DataInputStream counts as opening an InputStream and opening a DataOutputStream counts as opening an OutputStream. Trying to open another InputStream or OutputStream causes an IOException. Trying to open the InputStream or OutputStream after they have been closed causes an IOException.

The methods of StreamConnection are not synchronized. The only stream method that can be called safely in another thread is close.

6.4.6 javax.microedition.io.ConnectionNotFoundException

Declaration

```
@SCJAllowed
 public class ConnectionNotFoundException extends java.io.IOException
```

Description

This class is used to signal that a connection target cannot be found, or the protocol type is not supported.

Constructors

```
@SCJAllowed
@SCJMayAllocate({})
@SCJMaySelfSuspend(false)
@SCJPhase({
   javax.safetycritical.annotate.Phase.STARTUP,
   javax.safetycritical.annotate.Phase.INITIALIZATION,
   javax.safetycritical.annotate.Phase.RUN,
   javax.safetycritical.annotate.Phase.CLEANUP })
public ConnectionNotFoundException(String s)
```

Constructs a ConnectionNotFoundException with the specified detail message. A detail message is a String that describes this particular exception.

s — the detail message. If s is null, no detail message is provided.

```
@SCJAllowed
@SCJMayAllocate({})
@SCJMaySelfSuspend(false)
@SCJPhase({
    javax.safetycritical.annotate.Phase.STARTUP,
    javax.safetycritical.annotate.Phase.INITIALIZATION,
    javax.safetycritical.annotate.Phase.RUN,
    javax.safetycritical.annotate.Phase.CLEANUP })
public ConnectionNotFoundException( )
```

This constructor behaves the same as calling ConnectionNotFoundException(String) with the arguments (null).

6.4.7 javax.safetycritical.io.ConsoleConnection

Declaration

```
@SCJAllowed
 public class ConsoleConnection implements
     javax.microedition.io.StreamConnection extends java.lang.Object
```

Description

A connection for the default I/O device. The console connection can be obtained by the javax.microedition.io.Connector class with the openOutputStream method by providing ”console:” as the base url

Methods

```
@Override
@SCJAllowed
@SCJMaySelfSuspend(true)
@SCJMayAllocate({
    javax.safetycritical.annotate.AllocationContext.CURRENT,
    javax.safetycritical.annotate.AllocationContext.INNER,
    javax.safetycritical.annotate.AllocationContext.OUTER})
@SCJPhase({
    javax.safetycritical.annotate.Phase.STARTUP,
    javax.safetycritical.annotate.Phase.INITIALIZATION,
    javax.safetycritical.annotate.Phase.RUN,
    javax.safetycritical.annotate.Phase.CLEANUP })
public void close( )
```

Closes this console connection.

```
@Override
@SCJAllowed
@SCJMaySelfSuspend(false)
@SCJMayAllocate({
   javax.safetycritical.annotate.AllocationContext.CURRENT,
   javax.safetycritical.annotate.AllocationContext.INNER,
   javax.safetycritical.annotate.AllocationContext.OUTER})
@SCJPhase({
   javax.safetycritical.annotate.Phase.STARTUP,
   javax.safetycritical.annotate.Phase.INITIALIZATION,
   javax.safetycritical.annotate.Phase.RUN,
   javax.safetycritical.annotate.Phase.CLEANUP })
public java.io.InputStream openInputStream( )
```

returns the input stream for this console connection.

```
@Override
@SCJAllowed
@SCJMaySelfSuspend(false)
@SCJMayAllocate({
   javax.safetycritical.annotate.AllocationContext.CURRENT,
   javax.safetycritical.annotate.AllocationContext.INNER,
   javax.safetycritical.annotate.AllocationContext.OUTER})
@SCJPhase({
   javax.safetycritical.annotate.Phase.STARTUP,
   javax.safetycritical.annotate.Phase.INITIALIZATION,
   javax.safetycritical.annotate.Phase.RUN,
   javax.safetycritical.annotate.Phase.CLEANUP })
public java.io.OutputStream openOutputStream( )
```

returns the output stream for this console connection.

6.4.8 javax.safetycritical.io.ConnectionFactory

Declaration

```
@SCJAllowed
 public abstract class ConnectionFactory extends java.lang.Object
```

Description

A factory for creating user defined connections.

Constructors

```
@SCJAllowed
@SCJMayAllocate({})
```

```
@SCJMaySelfSuspend(false)
@SCJPhase({
    javax.safetycritical.annotate.Phase.STARTUP,
    javax.safetycritical.annotate.Phase.INITIALIZATION,
    javax.safetycritical.annotate.Phase.RUN,
    javax.safetycritical.annotate.Phase.CLEANUP })
protected ConnectionFactory(String name)
```

Create a connection factory.

name — Connection name used for connection request in Connector.

Methods

```
@SCJAllowed
@SCJMaySelfSuspend(false)
@SCJMayAllocate({
    javax.safetycritical.annotate.AllocationContext.CURRENT,
    javax.safetycritical.annotate.AllocationContext.INNER,
    javax.safetycritical.annotate.AllocationContext.OUTER})
@SCJPhase({
    javax.safetycritical.annotate.Phase.STARTUP,
    javax.safetycritical.annotate.Phase.INITIALIZATION,
    javax.safetycritical.annotate.Phase.RUN,
    javax.safetycritical.annotate.Phase.CLEANUP })
public abstract javax.microedition.io.Connection create(String url)
  throws
     java.io.IOException, javax.microedition.io.ConnectionNotFoundException
```

Create a connection for the URL type of this factory.

url — URL for which to create the connection.

returns a connection for the URL.

Throws IOException when some other I/O problem is encountered.

```
@SCJAllowed
@Override
@SCJMayAllocate({})
@SCJMaySelfSuspend(false)
@SCJPhase({
    javax.safetycritical.annotate.Phase.STARTUP,
    javax.safetycritical.annotate.Phase.INITIALIZATION,
    javax.safetycritical.annotate.Phase.RUN,
    javax.safetycritical.annotate.Phase.CLEANUP })
public boolean equals(Object other)
```

```
@SCJAllowed
@SCJMayAllocate({})
@SCJMaySelfSuspend(false)
@SCJPhase({
    javax.safetycritical.annotate.Phase.STARTUP,
    javax.safetycritical.annotate.Phase.INITIALIZATION,
    javax.safetycritical.annotate.Phase.RUN,
    javax.safetycritical.annotate.Phase.CLEANUP })
public static javax.safetycritical.io.ConnectionFactory getRegistered(String name)
```

Get a reference to the already registered factory for a given protocol.

name — The name of the connection type.

returns The ConnectionFactory associated with the name, or null if no ConnectionFactory is registered.

```
@SCJAllowed
@SCJMayAllocate({})
@SCJMaySelfSuspend(false)
@SCJPhase({
    javax.safetycritical.annotate.Phase.STARTUP,
    javax.safetycritical.annotate.Phase.INITIALIZATION,
    javax.safetycritical.annotate.Phase.RUN,
    javax.safetycritical.annotate.Phase.CLEANUP })
public final java.lang.String getServiceName( )
```

Return the service name for a connection factory.

returns service name.

```
@SCJAllowed
@SCJMaySelfSuspend(false)
@SCJMayAllocate({
    javax.safetycritical.annotate.AllocationContext.CURRENT,
    javax.safetycritical.annotate.AllocationContext.INNER,
    javax.safetycritical.annotate.AllocationContext.OUTER})
@SCJPhase({
    javax.safetycritical.annotate.Phase.STARTUP,
    javax.safetycritical.annotate.Phase.INITIALIZATION,
    javax.safetycritical.annotate.Phase.RUN,
    javax.safetycritical.annotate.Phase.CLEANUP })
public static void register(ConnectionFactory factory)
```

Register an application-defined connection type in the connection framework. The method getServiceName specifies the protocol a factory handles. When a factory is already registered for a given protocol, the new factory replaces the old one.

factory — the connection factory.

6.4.9 java.io.PrintStream

SCJ includes a simple print stream facility for use by the TCK or an application. This facility is derived from the CLDC version of a simple PrintStream to avoid introducing a special SCJ facility.

Declaration

```
@SCJAllowed
 public class PrintStream extends java.io.OutputStream
```

Description

A PrintStream adds functionality to an output stream, namely the ability to print representations of various data values conveniently. A PrintStream never throws an IOException; instead, exceptional situations merely set an internal flag that can be tested via the checkError method. Optionally, a PrintStream can be created to flush automatically; this means that the flush method is automatically invoked after a byte array is written, one of the println methods is invoked, or a newline character or byte ('\n') is written.

All characters printed by a PrintStream are converted into bytes using the platform's default character encoding.

Constructors

```
@SCJAllowed
@SCJMayAllocate({})
@SCJMaySelfSuspend(true)
@SCJPhase({
   javax.safetycritical.annotate.Phase.STARTUP,
   javax.safetycritical.annotate.Phase.INITIALIZATION,
   javax.safetycritical.annotate.Phase.RUN,
   javax.safetycritical.annotate.Phase.CLEANUP })
public PrintStream(OutputStream out)
```

Create a new print stream. This stream will not flush automatically.

out — The output stream to which values and objects will be printed.

Methods

```
@SCJAllowed
@SCJMaySelfSuspend(true)
@SCJMayAllocate({
   javax.safetycritical.annotate.AllocationContext.CURRENT,
   javax.safetycritical.annotate.AllocationContext.INNER,
   javax.safetycritical.annotate.AllocationContext.OUTER})
@SCJPhase({
```

```
    javax.safetycritical.annotate.Phase.STARTUP,
    javax.safetycritical.annotate.Phase.INITIALIZATION,
    javax.safetycritical.annotate.Phase.RUN,
    javax.safetycritical.annotate.Phase.CLEANUP })
public boolean checkError( )
```

Flush the stream and check its error state. The internal error state is set to true when the underlying output stream throws an IOException, and when the setError method is invoked.

returns true if and only if this stream has encountered an IOException, or the setError method has been invoked.

```
@SCJAllowed
@SCJMaySelfSuspend(true)
@SCJMayAllocate({
    javax.safetycritical.annotate.AllocationContext.CURRENT,
    javax.safetycritical.annotate.AllocationContext.INNER,
    javax.safetycritical.annotate.AllocationContext.OUTER})
@SCJPhase({
    javax.safetycritical.annotate.Phase.STARTUP,
    javax.safetycritical.annotate.Phase.INITIALIZATION,
    javax.safetycritical.annotate.Phase.RUN,
    javax.safetycritical.annotate.Phase.CLEANUP })
public void close( )
```

Close the stream. This is done by flushing the stream and then closing the underlying output stream.

See Also: java.io.OutputStream.close()

```
@SCJAllowed
@SCJMaySelfSuspend(true)
@SCJMayAllocate({
    javax.safetycritical.annotate.AllocationContext.CURRENT,
    javax.safetycritical.annotate.AllocationContext.INNER,
    javax.safetycritical.annotate.AllocationContext.OUTER})
@SCJPhase({
    javax.safetycritical.annotate.Phase.STARTUP,
    javax.safetycritical.annotate.Phase.INITIALIZATION,
    javax.safetycritical.annotate.Phase.RUN,
    javax.safetycritical.annotate.Phase.CLEANUP })
public void flush( )
```

Flush the stream. This is done by writing any buffered output bytes to the underlying output stream and then flushing that stream.

See Also: java.io.OutputStream.flush()

```
@SCJAllowed
@SCJMaySelfSuspend(false)
@SCJMayAllocate({
   javax.safetycritical.annotate.AllocationContext.CURRENT,
   javax.safetycritical.annotate.AllocationContext.INNER,
   javax.safetycritical.annotate.AllocationContext.OUTER})
@SCJPhase({
   javax.safetycritical.annotate.Phase.STARTUP,
   javax.safetycritical.annotate.Phase.INITIALIZATION,
   javax.safetycritical.annotate.Phase.RUN,
   javax.safetycritical.annotate.Phase.CLEANUP })
public void print(int i)
```

Print an integer. The string produced by java.lang.String.valueOf(i) is translated into bytes according to the platform's default character encoding, and these bytes are written in exactly the manner of the write() method.

i — The int to be printed.

See Also: java.lang.Integer.toString(int)

```
@SCJAllowed
@SCJMaySelfSuspend(false)
@SCJMayAllocate({
   javax.safetycritical.annotate.AllocationContext.CURRENT,
   javax.safetycritical.annotate.AllocationContext.INNER,
   javax.safetycritical.annotate.AllocationContext.OUTER})
@SCJPhase({
   javax.safetycritical.annotate.Phase.STARTUP,
   javax.safetycritical.annotate.Phase.INITIALIZATION,
   javax.safetycritical.annotate.Phase.RUN,
   javax.safetycritical.annotate.Phase.CLEANUP })
public void print(char [] s)
```

Print an array of characters. The characters are converted into bytes according to the platform's default character encoding, and these bytes are written in exactly the manner of the write() method.

s — The array of chars to be printed.

Throws NullPointerException If s is null

```
@SCJAllowed
@SCJMaySelfSuspend(false)
@SCJMayAllocate({
```

```
    javax.safetycritical.annotate.AllocationContext.CURRENT,
    javax.safetycritical.annotate.AllocationContext.INNER,
    javax.safetycritical.annotate.AllocationContext.OUTER})
@SCJPhase({
    javax.safetycritical.annotate.Phase.STARTUP,
    javax.safetycritical.annotate.Phase.INITIALIZATION,
    javax.safetycritical.annotate.Phase.RUN,
    javax.safetycritical.annotate.Phase.CLEANUP })
public void print(Object obj)
```

Print an object. The string produced by the java.lang.String.valueOf(obj) method is translated into bytes according to the platform's default character encoding, and these bytes are written in exactly the manner of the write() method.

obj — The Object to be printed.

See Also: java.lang.Object.toString()

```
@SCJAllowed
@SCJMaySelfSuspend(false)
@SCJMayAllocate({
    javax.safetycritical.annotate.AllocationContext.CURRENT,
    javax.safetycritical.annotate.AllocationContext.INNER,
    javax.safetycritical.annotate.AllocationContext.OUTER})
@SCJPhase({
    javax.safetycritical.annotate.Phase.STARTUP,
    javax.safetycritical.annotate.Phase.INITIALIZATION,
    javax.safetycritical.annotate.Phase.RUN,
    javax.safetycritical.annotate.Phase.CLEANUP })
public void print(String s)
```

Print a string. If the argument is null then the string "null" is printed. Otherwise, the string's characters are converted into bytes according to the platform's default character encoding, and these bytes are written in exactly the manner of the write() method.

s — The String to be printed.

```
@SCJAllowed
@SCJMaySelfSuspend(false)
@SCJMayAllocate({
    javax.safetycritical.annotate.AllocationContext.CURRENT,
    javax.safetycritical.annotate.AllocationContext.INNER,
    javax.safetycritical.annotate.AllocationContext.OUTER})
@SCJPhase({
    javax.safetycritical.annotate.Phase.STARTUP,
    javax.safetycritical.annotate.Phase.INITIALIZATION,
```

```
    javax.safetycritical.annotate.Phase.RUN,
    javax.safetycritical.annotate.Phase.CLEANUP })
public void print(long l)
```

Print a long integer. The string produced by java.lang.String.valueOf(l) is translated into bytes according to the platform's default character encoding, and these bytes are written in exactly the manner of the write() method.

l — The long to be printed.

See Also: java.lang.Long.toString(long)

```
@SCJAllowed
@SCJMaySelfSuspend(false)
@SCJMayAllocate({
    javax.safetycritical.annotate.AllocationContext.CURRENT,
    javax.safetycritical.annotate.AllocationContext.INNER,
    javax.safetycritical.annotate.AllocationContext.OUTER})
@SCJPhase({
    javax.safetycritical.annotate.Phase.STARTUP,
    javax.safetycritical.annotate.Phase.INITIALIZATION,
    javax.safetycritical.annotate.Phase.RUN,
    javax.safetycritical.annotate.Phase.CLEANUP })
public void print(char c)
```

Print a character. The character is translated into one or more bytes according to the platform's default character encoding, and these bytes are written in exactly the manner of the write() method.

c — The char to be printed.

```
@SCJAllowed
@SCJMaySelfSuspend(false)
@SCJMayAllocate({
    javax.safetycritical.annotate.AllocationContext.CURRENT,
    javax.safetycritical.annotate.AllocationContext.INNER,
    javax.safetycritical.annotate.AllocationContext.OUTER})
@SCJPhase({
    javax.safetycritical.annotate.Phase.STARTUP,
    javax.safetycritical.annotate.Phase.INITIALIZATION,
    javax.safetycritical.annotate.Phase.RUN,
    javax.safetycritical.annotate.Phase.CLEANUP })
public void print(boolean b)
```

Print a boolean value. The string produced by java.lang.String.valueOf(b) is translated into bytes according to the platform's default character encoding, and these bytes are written in exactly the manner of the write() method.

b — The boolean to be printed.

```
@SCJAllowed
@SCJMaySelfSuspend(true)
@SCJMayAllocate({
   javax.safetycritical.annotate.AllocationContext.CURRENT,
   javax.safetycritical.annotate.AllocationContext.INNER,
   javax.safetycritical.annotate.AllocationContext.OUTER})
@SCJPhase({
   javax.safetycritical.annotate.Phase.STARTUP,
   javax.safetycritical.annotate.Phase.INITIALIZATION,
   javax.safetycritical.annotate.Phase.RUN,
   javax.safetycritical.annotate.Phase.CLEANUP })
public void println(boolean x)
```

Print a boolean and then terminate the line. This method behaves as though it invokes print(x) and then println().

x — The boolean to be printed.

```
@SCJAllowed
@SCJMaySelfSuspend(true)
@SCJMayAllocate({
   javax.safetycritical.annotate.AllocationContext.CURRENT,
   javax.safetycritical.annotate.AllocationContext.INNER,
   javax.safetycritical.annotate.AllocationContext.OUTER})
@SCJPhase({
   javax.safetycritical.annotate.Phase.STARTUP,
   javax.safetycritical.annotate.Phase.INITIALIZATION,
   javax.safetycritical.annotate.Phase.RUN,
   javax.safetycritical.annotate.Phase.CLEANUP })
public void println(char x)
```

Print a character and then terminate the line. This method behaves as though it invokes print(x) and then println().

x — The char to be printed.

```
@SCJAllowed
@SCJMaySelfSuspend(true)
@SCJMayAllocate({
   javax.safetycritical.annotate.AllocationContext.CURRENT,
   javax.safetycritical.annotate.AllocationContext.INNER,
   javax.safetycritical.annotate.AllocationContext.OUTER})
@SCJPhase({
   javax.safetycritical.annotate.Phase.STARTUP,
   javax.safetycritical.annotate.Phase.INITIALIZATION,
```

```
    javax.safetycritical.annotate.Phase.RUN,
    javax.safetycritical.annotate.Phase.CLEANUP })
public void println(int x)
```

Print an integer and then terminate the line. This method behaves as though it invokes print(x) and then println().

x — The int to be printed.

```
@SCJAllowed
@SCJMaySelfSuspend(true)
@SCJMayAllocate({
    javax.safetycritical.annotate.AllocationContext.CURRENT,
    javax.safetycritical.annotate.AllocationContext.INNER,
    javax.safetycritical.annotate.AllocationContext.OUTER})
@SCJPhase({
    javax.safetycritical.annotate.Phase.STARTUP,
    javax.safetycritical.annotate.Phase.INITIALIZATION,
    javax.safetycritical.annotate.Phase.RUN,
    javax.safetycritical.annotate.Phase.CLEANUP })
public void println(char [] x)
```

Print an array of characters and then terminate the line. This method behaves as though it invokes print(x) and then println().

x — an array of chars to print.

```
@SCJAllowed
@SCJMaySelfSuspend(true)
@SCJMayAllocate({
    javax.safetycritical.annotate.AllocationContext.CURRENT,
    javax.safetycritical.annotate.AllocationContext.INNER,
    javax.safetycritical.annotate.AllocationContext.OUTER})
@SCJPhase({
    javax.safetycritical.annotate.Phase.STARTUP,
    javax.safetycritical.annotate.Phase.INITIALIZATION,
    javax.safetycritical.annotate.Phase.RUN,
    javax.safetycritical.annotate.Phase.CLEANUP })
public void println(String x)
```

Print a String and then terminate the line. This method behaves as though it invokes print(x) and then println().

x — The String to be printed.

```
@SCJAllowed
@SCJMaySelfSuspend(true)
@SCJMayAllocate({
   javax.safetycritical.annotate.AllocationContext.CURRENT,
   javax.safetycritical.annotate.AllocationContext.INNER,
   javax.safetycritical.annotate.AllocationContext.OUTER})
@SCJPhase({
   javax.safetycritical.annotate.Phase.STARTUP,
   javax.safetycritical.annotate.Phase.INITIALIZATION,
   javax.safetycritical.annotate.Phase.RUN,
   javax.safetycritical.annotate.Phase.CLEANUP })
public void println(Object x)
```

Print an Object and then terminate the line. This method behaves as though it invokes print(x) and then println().

x — The Object to be printed.

```
@SCJAllowed
@SCJMaySelfSuspend(true)
@SCJMayAllocate({
   javax.safetycritical.annotate.AllocationContext.CURRENT,
   javax.safetycritical.annotate.AllocationContext.INNER,
   javax.safetycritical.annotate.AllocationContext.OUTER})
@SCJPhase({
   javax.safetycritical.annotate.Phase.STARTUP,
   javax.safetycritical.annotate.Phase.INITIALIZATION,
   javax.safetycritical.annotate.Phase.RUN,
   javax.safetycritical.annotate.Phase.CLEANUP })
public void println(long x)
```

Print a long and then terminate the line. This method behaves as though it invokes print(x) and then println().

x — a The long to be printed.

```
@SCJAllowed
@SCJMaySelfSuspend(true)
@SCJMayAllocate({
   javax.safetycritical.annotate.AllocationContext.CURRENT,
   javax.safetycritical.annotate.AllocationContext.INNER,
   javax.safetycritical.annotate.AllocationContext.OUTER})
@SCJPhase({
   javax.safetycritical.annotate.Phase.STARTUP,
   javax.safetycritical.annotate.Phase.INITIALIZATION,
   javax.safetycritical.annotate.Phase.RUN,
   javax.safetycritical.annotate.Phase.CLEANUP })
public void println( )
```

Terminate the current line by writing the line separator string. The line separator string is defined by the system property line.separator, and is not necessarily a single newline character ('\n').

```
@SCJAllowed
@SCJMayAllocate({})
@SCJMaySelfSuspend(false)
@SCJPhase({
   javax.safetycritical.annotate.Phase.STARTUP,
   javax.safetycritical.annotate.Phase.INITIALIZATION,
   javax.safetycritical.annotate.Phase.RUN,
   javax.safetycritical.annotate.Phase.CLEANUP })
protected void setError( )
```

Set the error state of the stream to true.

Since
JDK1.1

```
@SCJAllowed
@SCJMaySelfSuspend(true)
@SCJMayAllocate({
   javax.safetycritical.annotate.AllocationContext.CURRENT,
   javax.safetycritical.annotate.AllocationContext.INNER,
   javax.safetycritical.annotate.AllocationContext.OUTER})
@SCJPhase({
   javax.safetycritical.annotate.Phase.STARTUP,
   javax.safetycritical.annotate.Phase.INITIALIZATION,
   javax.safetycritical.annotate.Phase.RUN,
   javax.safetycritical.annotate.Phase.CLEANUP })
public void write(byte [] buf, int off, int len)
```

Write len bytes from the specified byte array starting at offset off to this stream. If automatic flushing is enabled then the flush method will be invoked.

Note that the bytes will be written as given; to write characters that will be translated according to the platform's default character encoding, use the print() or println() methods.

buf — A byte array.

off — Offset from which to start taking bytes.

len — Number of bytes to write.

```
@SCJAllowed
@SCJMaySelfSuspend(true)
@SCJMayAllocate({
   javax.safetycritical.annotate.AllocationContext.CURRENT,
   javax.safetycritical.annotate.AllocationContext.INNER,
   javax.safetycritical.annotate.AllocationContext.OUTER})
@SCJPhase({
   javax.safetycritical.annotate.Phase.STARTUP,
   javax.safetycritical.annotate.Phase.INITIALIZATION,
   javax.safetycritical.annotate.Phase.RUN,
   javax.safetycritical.annotate.Phase.CLEANUP })
public void write(int b)
```

Write the specified byte to this stream. If the byte is a newline and automatic flushing is enabled then the flush method will be invoked.

Note that the byte is written as given; to write a character that will be translated according to the platform's default character encoding, use the print() or println() methods.

b — The byte to be written.

See Also: java.io.PrintStream.print(char), java.io.PrintStream.println(char)

6.5 Rationale

In the creation of SCJ, it was determined that the standard Java I/O classes (e.g., in packages java.io, java.net, java.file, and java.nio) would require too many classes that are not compatible with a safety-critical application. In contrast, the basic mechanism of the connection classes, as defined by Java Micro Edition, provides a lightweight framework for stream based I/O.

To provide a minimal, standard way to communicate simple text messages, the Java Micro Edition console connection is subsetted. This connection provides the ability to report the test results of the TCK on all compliant SCJ implementations.

To simplify the conversion between Java strings, which are based on Unicode, and the binary based connection classes, a simplified version of the CLDC's PrintStream is provided.

6.6 Compatibility

These SCJ I/O classes use the Java Micro Edition connection framework. A SCJ implementation shall support the console connection. All other Micro Edition con-

nection types are optional. Application-provided connections can be registered with a factory class provided by SCJ.

The Java Micro Edition (J2ME) connection framework is compatible with RTSJ, which itself is based on the CLDC specification of J2ME. The factory for user implemented connections is not available in the RTSJ.

Chapter 7

Memory Management

7.1 Overview

As with the RTSJ, every object allocation performed by an SCJ application is associated with a particular *allocation context*. Each allocation context represents a finite amount of allocatable memory. In both the SCJ and RTSJ, application code explicitly controls which allocation context is current for each schedulable object. Application programs change the current allocation context by invoking special infrastructure methods (e.g. ManagedMemory.enterPrivateMemory or ManagedMemory.executeInAreaOf with the SCJ), passing a Runnable argument for which the infrastructure will invoke the run method after arranging for the newly selected allocation context to be treated as the current allocation context.

As long as any active call chain includes methods associated with a particular allocation context, that allocation context is considered to be live, and all of the objects it contains are retained. When no schedulable objects are executing methods associated with a particular allocation context, all of the memory for objects contained within that allocation context is reclaimed before any schedulable object is allowed to enter (or re-enter) that allocation context. One difference between the RTSJ and SCJ is that the latter prohibits object finalizers. Thus, an SCJ infrastructure is able to reclaim all of the memory associated with an unused allocation area in constant time.

The RTSJ defines a variety of allocation contexts, including HeapMemory, ImmortalMemory, and various kinds of ScopedMemory. SCJ is much more restrictive. It supports only three concrete allocation area types: ImmortalMemory, MissionMemory, and PrivateMemory. To abstract common functionality, both MissionMemory and PrivateMemory extend ManagedMemory, an abstract subclass of the RTSJ's StackedMemory class.

The Overview and Rationale sections are not normative but are provided to improve understanding of the normative sections. All of the other sections of this chapter are normative.

7.2 Semantics and Requirements

As discussed in Chapter 3, SCJ supports the notion of a mission and a mission life cycle. An SCJ mission has four phases as shown in Figure 3.1: start up, initialization, execution, and clean up.

Objects needed for a given mission are allocated in a special allocation context called *mission memory*. Mission memory remains active for the duration of the mission and acts like an immortal memory for that mission. Normally, allocation of objects in mission memory takes place in the initialization phase and those objects persist throughout the life of the mission. Temporary objects may be allocated in memory areas private to a schedulable object during the execution phase. These areas are instances of PrivateMemory. Nested missions are supported, so an application may have more than one active mission memory.

Both MissionMemory and PrivateMemory are direct subclasses of ManagedMemory that provide a means for the infrastructure to track its scoped memory areas. General memory management static methods can be found in ManagedMemory as well.

In SCJ, each schedulable object can allocate objects in its own private scoped memory areas. As with the RTSJ, the term *backing store* is used to represent the location in memory where the space for objects allocated in these memory areas is taken.

7.2.1 Memory Model

The following defines the requirements for the SCJ memory model that enables object creation without requiring garbage collection, avoiding memory fragmentation, and without a need to explicitly free memory:

- Only linear-time scoped memory and the immortal memory areas shall be supported. Variable time scoped memory and heap memory areas are not supported.
- A MissionMemory object shall be provided; it shall be entered before the mission initialization phase and exited after the mission clean up phase.
- Objects allocated in mission memory shall not be reclaimed throughout the duration of a given mission.
- For an event handler, all allocations performed during a release (see Chapter 4) shall, by default, be performed in a private memory. The memory allocated

to objects created in this private memory shall be reclaimed at the end of the release.

- For a thread, allocations performed during its execution (see Chapter 4) shall, by default, be performed in its private memory. The memory allocated to objects created in this private memory shall be reclaimed when the thread's run() method terminates.
- Schedulable objects may create and enter into nested private memory areas. These memory areas shall not be shared with other schedulable objects and shall be entered directly from the private memory in which they are created. The constructor of PrivateMemory is not visible to the application.
- Backing store shall be managed as specified in the ScopeParameters provided to schedulable objects.
  - The backing store for a private memory shall be taken from the backing store reservation of its owning schedulable object.
  - The backing store for mission memory shall be taken from the backing store reservation of its mission sequencer.
- SCJ shall not support object finalizers. A similar effect can be obtained for
  - mission memory — by using the Mission.cleanUp method;
  - the per-release memory area of a managed event handler — by encapsulating the code of the handler's handleAsyncEvent method in a try statement that includes a finally clause;
  - the per-release memory area of a managed thread — by encapsulating the code of the thread's run method in a try statement that includes a finally clause;
  - a nested private memory area — by encapsulating the code of the run method passed to ManagedMemory.enterPrivateMemory in a try statement that includes a finally clause.
- SCJ shall conform to the Java memory model. In addition, all access to raw memory is considered to be volatile access (see Section 5.2).

Figure 7.1 illustrates the use of hierarchical memory areas within SCJ. The diagram shows the scope stacks for five schedulable objects (SO A .. E). They all share immortal and mission memory at their base.

7.3 Level Considerations

All schedulable objects at all compliance levels are able to use private memory areas for the storage of temporary objects. The scheduling approach adopted at each level, however, does have an impact on how the memory areas and their associated backing storage are managed.

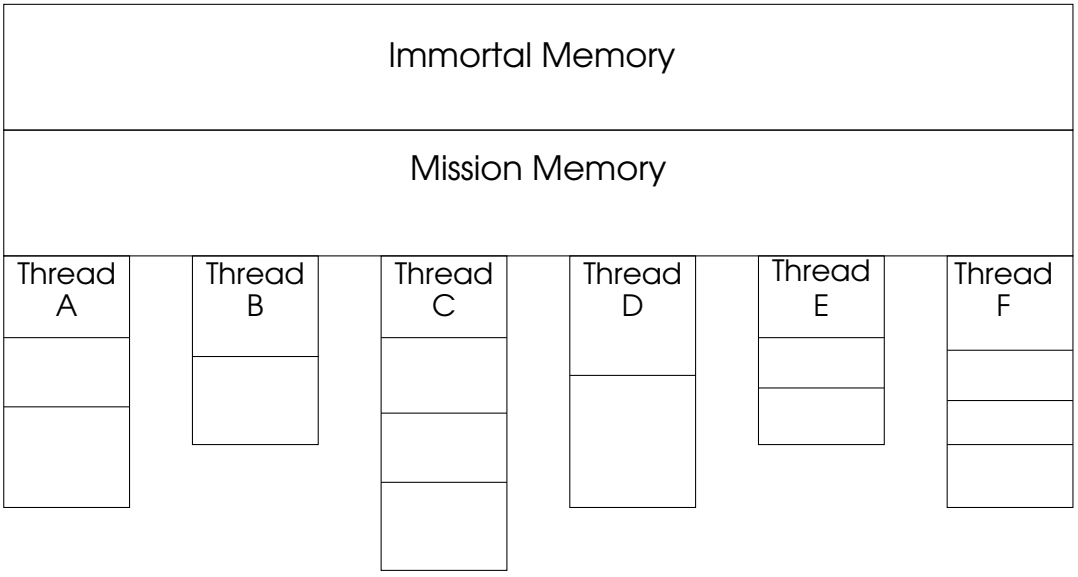

Figure 7.1: Example of Memory Areas used by a Level 1 Application

7.3.1 Level 0

Level 0 supports a single mission sequencer. The same mission memory shall be reused for each mission in the sequence; however, the size of the mission memory may be changed between missions. Memory used by objects created inside mission memory during one mission shall be reclaimed after the termination of the mission. Each PeriodicEventHandler has a PrivateMemory that is entered for the duration of its handleAsyncEvent method called within its frame. This corresponds to release and completion in higher SCJ compliance levels. The application programmer may enter additional PrivateMemory areas within a frame, so simple nesting of private memory is possible.

Since no two PeriodicEventHandlers in a Level 0 application are permitted to execute simultaneously, the backing store for the private memories may be reused. As a consequence, the total size required can be the maximum of the backing store sizes needed for each handler's private memories. In order for this to be achieved, the implementation may revoke the backing store reservation for the private memory of a periodic event handler at the end of its release.

7.3.2 Level 1

Level 1 supports a sequence of missions and private memory for each handler as well, but the handlers are run asynchronously. Level 1 shall have the same memory

semantics as Level 0, except the backing store reservation for each handler shall remain in place for the entire mission.

7.3.3 Level 2

Level 2 shall have the same memory semantics as Level 1 with the addition of support for nested mission memories. A nested mission memory is created when its associated nested mission sequencer is created. A nested MissionSequencer can be created only during execution of the new mission's initialize method.

7.4 Memory-Related APIs

SCJ supports only a subset of the RTSJ memory model. Consequently many of the methods are absent (and, therefore the complexity of the overall model is reduced). The application can only create SCJ-defined private memory areas. Figure 7.2 provides an overview of the supported interfaces and classes.

7.4.1 Class javax.realtime.MemoryParameters

Refer to Section 4.4.6

7.4.2 Class javax.realtime.MemoryArea

Declaration

```
@SCJAllowed
 public abstract class MemoryArea extends java.lang.Object
```

Description

All allocation contexts are implemented by memory areas. This is the base class for all memory areas.

Methods

```
@SCJAllowed
@SCJMaySelfSuspend(false)
@SCJMayAllocate({
   javax.safetycritical.annotate.AllocationContext.CURRENT,
   javax.safetycritical.annotate.AllocationContext.INNER,
   javax.safetycritical.annotate.AllocationContext.OUTER})
@SCJPhase({
```

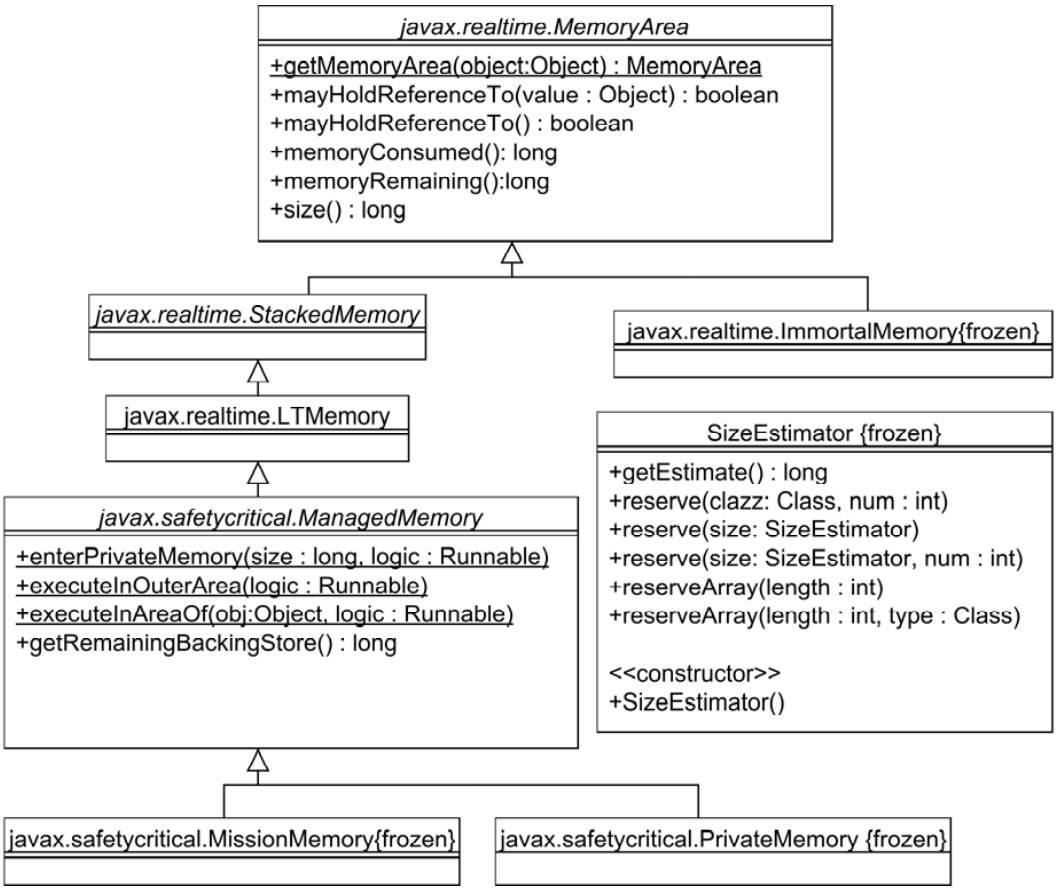

Figure 7.2: Overview of MemoryArea-Related Classes

```
    javax.safetycritical.annotate.Phase.STARTUP,
    javax.safetycritical.annotate.Phase.INITIALIZATION,
    javax.safetycritical.annotate.Phase.RUN,
    javax.safetycritical.annotate.Phase.CLEANUP })
public static javax.realtime.MemoryArea getMemoryArea(Object object)
```

Get the memory area in which object is allocated,

returns the memory area

```
@SCJAllowed
@SCJMaySelfSuspend(false)
@SCJMayAllocate({
    javax.safetycritical.annotate.AllocationContext.CURRENT,
    javax.safetycritical.annotate.AllocationContext.INNER,
    javax.safetycritical.annotate.AllocationContext.OUTER})
@SCJPhase({
    javax.safetycritical.annotate.Phase.STARTUP,
    javax.safetycritical.annotate.Phase.INITIALIZATION,
    javax.safetycritical.annotate.Phase.RUN,
    javax.safetycritical.annotate.Phase.CLEANUP })
public boolean mayHoldReferenceTo(Object value)
```

Determine whether an object allocated in the memory area represented by this can hold a reference to the object value.

returns true when value can be assigned to a field of an object in this memory area, otherwise false.

```
@SCJAllowed
@SCJMaySelfSuspend(false)
@SCJMayAllocate({
    javax.safetycritical.annotate.AllocationContext.CURRENT,
    javax.safetycritical.annotate.AllocationContext.INNER,
    javax.safetycritical.annotate.AllocationContext.OUTER})
@SCJPhase({
    javax.safetycritical.annotate.Phase.STARTUP,
    javax.safetycritical.annotate.Phase.INITIALIZATION,
    javax.safetycritical.annotate.Phase.RUN,
    javax.safetycritical.annotate.Phase.CLEANUP })
public boolean mayHoldReferenceTo( )
```

Determine whether an object allocated in the memory area represented by this can hold a reference to an object allocated in the current memory area.

returns true when an object in the current memory area can be assigned to a field of an object in this memory area, otherwise false.

```
@SCJAllowed
@SCJMaySelfSuspend(false)
@SCJMayAllocate({
   javax.safetycritical.annotate.AllocationContext.CURRENT,
   javax.safetycritical.annotate.AllocationContext.INNER,
   javax.safetycritical.annotate.AllocationContext.OUTER})
@SCJPhase({
   javax.safetycritical.annotate.Phase.STARTUP,
   javax.safetycritical.annotate.Phase.INITIALIZATION,
   javax.safetycritical.annotate.Phase.RUN,
   javax.safetycritical.annotate.Phase.CLEANUP })
public long size( )
```

The size of a memory area is memoryConsumed() + memoryRemaining().

returns the total size of this memory area.

7.4.3 Class javax.realtime.ImmortalMemory

Declaration

```
@SCJAllowed
 public final class ImmortalMemory extends javax.realtime.MemoryArea
```

Description

This class represents immortal memory. Objects allocated in immortal memory are never reclaimed during the lifetime of the application. The singleton instance of this class is created and managed by the infrastructure, so no application visible constructors or methods are provided.

7.4.4 Class javax.realtime.memory.ScopedMemory

Declaration

```
@SCJAllowed
 public abstract class ScopedMemory extends javax.realtime.MemoryArea
```

Description

Scoped memory implements the scoped allocation context.

Methods

```
@SCJAllowed
@SCJMaySelfSuspend(false)
@SCJPhase({
   javax.safetycritical.annotate.Phase.INITIALIZATION,
```

```
    javax.safetycritical.annotate.Phase.RUN,
    javax.safetycritical.annotate.Phase.CLEANUP})
@SCJMayAllocate({})
public long backingStoreConsumed( )
```

Determines the amount of backing store consumed by this scoped memory and its children.

returns the total amount of backing store.

```
@SCJAllowed
@SCJMaySelfSuspend(false)
@SCJPhase({
    javax.safetycritical.annotate.Phase.INITIALIZATION,
    javax.safetycritical.annotate.Phase.RUN,
    javax.safetycritical.annotate.Phase.CLEANUP})
@SCJMayAllocate({})
public long backingStoreRemaining( )
```

Determines the remaining amount of backing store available to this scoped memory and its children.

returns the total amount of backing store.

```
@SCJAllowed
@SCJMaySelfSuspend(false)
@SCJPhase({
    javax.safetycritical.annotate.Phase.INITIALIZATION,
    javax.safetycritical.annotate.Phase.RUN,
    javax.safetycritical.annotate.Phase.CLEANUP})
@SCJMayAllocate({})
public long backingStoreSize( )
```

Determines the total amount of backing store for this scoped memory and its children.

returns the total amount of backing store.

7.4.5 Class javax.realtime.memory.ScopeParamters

Declaration

```
@SCJAllowed
 public class ScopeParameters extends javax.realtime.MemoryParameters
```

See Also: javax.realtime.MemoryParameters

Description

Extend memory parameters to provide limits for scoped memory.

Constructors

```
@SCJAllowed
@SCJMayAllocate({
   javax.safetycritical.annotate.AllocationContext.CURRENT,
   javax.safetycritical.annotate.AllocationContext.INNER,
   javax.safetycritical.annotate.AllocationContext.OUTER})
@SCJMaySelfSuspend(true)
@SCJPhase({
   javax.safetycritical.annotate.Phase.STARTUP,
   javax.safetycritical.annotate.Phase.INITIALIZATION,
   javax.safetycritical.annotate.Phase.RUN,
   javax.safetycritical.annotate.Phase.CLEANUP })
public ScopeParameters(long maxInitialArea,
  long maxImmortal,
  long maxContainingArea,
  long maxInitialBackingStore)
  throws java.lang.IllegalArgumentException
```

Create a ScopeParameters instance with the given values.

maxInitialArea — a limit on the amount of memory the schedulable may allocate in its initial scoped memory area. Units are in bytes. When zero, no allocation is allowed in the memory area. When the initial memory area is not a ScopedMemory , this parameter has no effect. To specify no limit, use MemoryParameters.UNLIMITED .

maxImmortal — A limit on the amount of memory the schedulable may allocate in the immortal area. Units are in bytes. When zero, no allocation allowed in immortal. To specify no limit, use MemoryParameters.UNLIMITED .

maxContainingArea — a limit on the amount of memory the schedulable may allocate in memory area where it was created. Units are in bytes. When zero, no allocation is allowed in the memory area. When the containing memory area is not a ScopedMemory , this parameter has no effect. To specify no limit, use MemoryParameters.UNLIMITED . For schedulables created within a mission, the containing memory area is Mission memory. For the initial MissionSequencer, the initial memory area is Immortal memory.

maxInitialBackingStore — A limit on the amount of backing store this task may allocate from backing store of its inital area, when that is a stacked memory. Units are in bytes. When zero, no allocation is allowed in that memory area. Backing store that is returned to the global backing store is subtracted from the limit. To specify no limit, use MemoryParameters.UNLIMITED .

Throws IllegalArgumentException when any value other than positive, zero, or **javax.realtime.MemoryParametersUNREFERENCED** is passed as the value of maxInitialArea , maxImmortal , maxParentBackingStore , or maxContainingArea.

```
@SCJAllowed
@SCJMayAllocate({
   javax.safetycritical.annotate.AllocationContext.CURRENT,
   javax.safetycritical.annotate.AllocationContext.INNER,
   javax.safetycritical.annotate.AllocationContext.OUTER})
@SCJMaySelfSuspend(true)
@SCJPhase({
   javax.safetycritical.annotate.Phase.STARTUP,
   javax.safetycritical.annotate.Phase.INITIALIZATION,
   javax.safetycritical.annotate.Phase.RUN,
   javax.safetycritical.annotate.Phase.CLEANUP })
public ScopeParameters(long maxInitialArea,
  long maxImmortal,
  long maxInitialBackingStore)
  throws java.lang.IllegalArgumentException
```

Same as ScopeParameters(maxInitialArea, maxImmortal, maxParentBackingStore, MemoryParameters.UNLIMITED).

Methods

```
@SCJAllowed
@SCJMaySelfSuspend(false)
@SCJPhase({
   javax.safetycritical.annotate.Phase.INITIALIZATION,
   javax.safetycritical.annotate.Phase.RUN,
   javax.safetycritical.annotate.Phase.CLEANUP})
@SCJMayAllocate({})
public long getMaxBackingStore( )
```

Determine the limit on backing store for this task.

returns the limit on backing store.

```
@SCJMaySelfSuspend(false)
@SCJPhase({
   javax.safetycritical.annotate.Phase.INITIALIZATION,
   javax.safetycritical.annotate.Phase.RUN,
   javax.safetycritical.annotate.Phase.CLEANUP})
@SCJMayAllocate({})
@SCJAllowed
public long getMaxContainingArea( )
```

Determine the limit on allocation in the area where the task was created.

returns the limit on allocation in the area where the task was created.

7.4.6 Class javax.realtime.memory.StackedMemory

Declaration

```
@SCJAllowed
 public class StackedMemory extends javax.realtime.memory.ScopedMemory
```

Description

This class can not be instantiated in SCJ. It is subclassed by MissionMemory and PrivateMemory. It has no visible methods for SCJ applications.

7.4.7 Class javax.safetycritical.ManagedMemory

Declaration

```
@SCJAllowed
 public abstract class ManagedMemory extends javax.realtime.memory.StackedMemory
```

Description

This is the base class for all safety-critical Java memory areas. This class is used by the SCJ infrastructure to manage all SCJ memory areas. This class has no constructors, so it cannot be extended by an application.

Methods

```
@SCJAllowed
@SCJMayAllocate({javax.safetycritical.annotate.AllocationContext.CURRENT})
@SCJMaySelfSuspend(false)
@SCJPhase({
   javax.safetycritical.annotate.Phase.STARTUP,
   javax.safetycritical.annotate.Phase.INITIALIZATION,
   javax.safetycritical.annotate.Phase.RUN,
   javax.safetycritical.annotate.Phase.CLEANUP })
public long backingStoreRemaining( )
```

This method determines the available memory for new objects in the current ManagedMemory area.

returns the size in bytes of the remaining available memory to in the Managed-Memory area.

```
@SCJAllowed
@SCJMaySelfSuspend(false)
@SCJPhase({
   javax.safetycritical.annotate.Phase.INITIALIZATION,
   javax.safetycritical.annotate.Phase.RUN,
   javax.safetycritical.annotate.Phase.CLEANUP})
```

```
@SCJMayAllocate({javax.safetycritical.annotate.AllocationContext.CURRENT})
public static void enterPrivateMemory(long size, Runnable logic)
  throws java.lang.IllegalStateException
```

Invoke the run method of logic with a fresh private memory area that is immediately nested within the current ManagedMemory area, sized to provide size bytes of allocatable memory as the current allocation area. Each instance of ManagedMemory maintains at most one inner-nested private memory area. In the case that enterPrivateMemory is invoked multiple times from within a particular ManagedMemory area without exiting that area, the first invocation instantiates the inner-nested private memory area and subsequent invocations resize and reuse the previously allocated private memory area. This is different from the case in which enterPrivateMemory is invoked from within a newly entered inner-nested PrivateMemory area. In this case, invocation of enterPrivateMemory results in creation and sizing of a new inner-nested private memory area.

size — is the number of allocatable memory bytes for the inner-nested private memory area.

logic — provides the run method that is to be executed within the inner-nested private memory area.

Throws IllegalStateException if the current allocation area is not the top-most (most recently entered) scope for the current schedulable object. (This would happen, for example, if the current schedulable object is in an outer-nested context as a result of having invoked, for example, executeInAreaOf).

Throws OutOfMemoryError if the currently running thread lacks sufficient backing store to have an inner-nested private memory area with size allocatable bytes; or if this is the first invocation of enterPrivateMemory from within the current allocation area and the current allocation area lacks sufficient backing store to allocate the inner-nested private memory area object.

```
@SCJAllowed
@SCJMayAllocate({})
@SCJPhase({
   javax.safetycritical.annotate.Phase.INITIALIZATION,
   javax.safetycritical.annotate.Phase.RUN,
   javax.safetycritical.annotate.Phase.CLEANUP})
@SCJMaySelfSuspend(false)
public static void executeInAreaOf(Object obj, Runnable logic)
```

Change the allocation context to the outer memory area where the object obj is allocated and invoke the run method of the logic Runnable.

obj — is the object allocated in the memory area that is entered.

logic — is the code to be executed in the entered memory area.

```
@SCJAllowed
@SCJMayAllocate({})
@SCJMaySelfSuspend(false)
@SCJPhase({
    javax.safetycritical.annotate.Phase.INITIALIZATION,
    javax.safetycritical.annotate.Phase.RUN,
    javax.safetycritical.annotate.Phase.CLEANUP})
public static void executeInOuterArea(Runnable logic)
```

Change the allocation context to the immediate outer memory area and invoke the run method of the Runnable.

logic — is the code to be executed in the entered memory area.

Throws IllegalStateException if the current memory area is ImmortalMemory.

7.4.8 Class javax.realtime.SizeEstimator

Declaration

```
@SCJAllowed
 public final class SizeEstimator extends java.lang.Object
```

Description

This class maintains a conservative upper bound of the amount of memory required to store a set of objects.

Many objects allocate other objects when they are constructed. SizeEstimator only estimates the memory requirement of the object itself; it does not include memory required for any objects allocated at construction time. If the Java implementation allocates a single Java object in several parts not separately visible to the application (if, for example, the object and its monitor are separate), the size estimate shall include the sum of the sizes of all the invisible parts that are allocated from the same memory area as the object.

Alignment considerations, and possibly other order-dependent issues may cause the allocator to leave a small amount of unusable space. Consequently, the size estimate cannot be seen as more than a close estimate, but SCJ requires that the size estimate shall represent a tight upper bound.

Constructors

```
@SCJAllowed
@SCJPhase({
    javax.safetycritical.annotate.Phase.INITIALIZATION,
    javax.safetycritical.annotate.Phase.RUN,
    javax.safetycritical.annotate.Phase.CLEANUP})
@SCJMaySelfSuspend(false)
@SCJMayAllocate({
    javax.safetycritical.annotate.AllocationContext.CURRENT,
    javax.safetycritical.annotate.AllocationContext.INNER,
    javax.safetycritical.annotate.AllocationContext.OUTER})
public SizeEstimator( )
```

Creates a new SizeEstimator object in the current allocation context.

Methods

```
@SCJAllowed
@SCJPhase({
    javax.safetycritical.annotate.Phase.INITIALIZATION,
    javax.safetycritical.annotate.Phase.RUN,
    javax.safetycritical.annotate.Phase.CLEANUP})
@SCJMayAllocate({})
@SCJMaySelfSuspend(false)
public void clear( )
```

Return the estimate to zero for reuse.

```
@SCJAllowed
@SCJPhase({
    javax.safetycritical.annotate.Phase.INITIALIZATION,
    javax.safetycritical.annotate.Phase.RUN,
    javax.safetycritical.annotate.Phase.CLEANUP})
@SCJMayAllocate({})
@SCJMaySelfSuspend(false)
public long getEstimate( )
```

Gets an estimate of the number of bytes needed to store all the objects reserved.

returns the estimated size in bytes.

```
@SCJAllowed
@SCJPhase({
    javax.safetycritical.annotate.Phase.INITIALIZATION,
    javax.safetycritical.annotate.Phase.RUN,
    javax.safetycritical.annotate.Phase.CLEANUP})
@SCJMayAllocate({})
@SCJMaySelfSuspend(false)
public void reserve(SizeEstimator size, int num)
```

Adds num times the value returned by size.getEstimate to the currently computed size of the set of reserved objects.

size — is the size.SizeEstimator whose size is to be reserved.

num — is the number of times to reserve this amount.

Throws IllegalArgumentException if size is null or num is negative.

```
@SCJAllowed
@SCJPhase({
   javax.safetycritical.annotate.Phase.INITIALIZATION,
   javax.safetycritical.annotate.Phase.RUN,
   javax.safetycritical.annotate.Phase.CLEANUP})
@SCJMayAllocate({})
@SCJMaySelfSuspend(false)
public void reserve(SizeEstimator size)
```

Adds the value returned by size.getEstimate to the currently computed size of the set of reserved objects.

size — is the size.SizeEstimator whose size is to be reserved.

Throws IllegalArgumentException if size is null.

```
@SCJAllowed
@SCJPhase({
   javax.safetycritical.annotate.Phase.INITIALIZATION,
   javax.safetycritical.annotate.Phase.RUN,
   javax.safetycritical.annotate.Phase.CLEANUP})
@SCJMayAllocate({})
@SCJMaySelfSuspend(false)
public void reserve(Class<?> clss, int num)
```

Adds the required memory size of num instances of a clss object to the currently computed size of the set of reserved objects.

clss — is the class to take into account.

num — is the number of instances of clss to estimate.

Throws IllegalArgumentException if clss is null or num is negative.

```
@SCJAllowed
@SCJPhase({
   javax.safetycritical.annotate.Phase.INITIALIZATION,
   javax.safetycritical.annotate.Phase.RUN,
   javax.safetycritical.annotate.Phase.CLEANUP})
@SCJMayAllocate({})
@SCJMaySelfSuspend(false)
public void reserveArray(int length, Class<?> type)
```

Adds the required memory size of an additional instance of an array of length primitive values of Class type to the currently computed size of the set of reserved objects. Class values for the primitive types shall be chosen from primitive class types such as Integer.TYPE, and Float.TYPE. The reservation shall leave room for an array of length of the primitive type corresponding to type.

length — is the number of entries in the array.

type — is the class representing a primitive type.

Throws IllegalArgumentException if length is negative, or type does not represent a primitive type.

```
@SCJAllowed
@SCJPhase({
   javax.safetycritical.annotate.Phase.INITIALIZATION,
   javax.safetycritical.annotate.Phase.RUN,
   javax.safetycritical.annotate.Phase.CLEANUP})
@SCJMayAllocate({})
@SCJMaySelfSuspend(false)
public void reserveArray(int length)
```

Adds the size of an instance of an array of length reference values to the currently computed size of the set of reserved objects.

length — is the number of entries in the array.

Throws IllegalArgumentException if length is negative.

```
@SCJAllowed
@SCJPhase({
   javax.safetycritical.annotate.Phase.INITIALIZATION,
   javax.safetycritical.annotate.Phase.RUN,
   javax.safetycritical.annotate.Phase.CLEANUP})
@SCJMayAllocate({})
@SCJMaySelfSuspend(false)
public void reserveLambda( )
```

Determine the size of a lambda with no closure and add it to this size estimator.

```
@SCJAllowed
@SCJPhase({
   javax.safetycritical.annotate.Phase.INITIALIZATION,
   javax.safetycritical.annotate.Phase.RUN,
   javax.safetycritical.annotate.Phase.CLEANUP})
@SCJMayAllocate({})
@SCJMaySelfSuspend(false)
public void reserveLambda(EnclosedType first, EnclosedType second)
```

Determine the size of a lambda with two variables in its closure and add it to this size estimator.

first — Type of first variable in closure.

second — Type of second variable in closure.

```
@SCJAllowed
@SCJPhase({
   javax.safetycritical.annotate.Phase.INITIALIZATION,
   javax.safetycritical.annotate.Phase.RUN,
   javax.safetycritical.annotate.Phase.CLEANUP})
@SCJMayAllocate({})
@SCJMaySelfSuspend(false)
public void reserveLambda(EnclosedType first,
  EnclosedType second,
  EnclosedType [] others)
```

Determine the size of a lambda with more than two variables in its closure and add it to this size estimator.

first — Type of first variable in closure.

second — Type of second variable in closure.

others — Types of additional variables in closure.

7.5 Rationale

Traditionally, safety-critical applications allocate all their data structures before the execution phase of the application begins. As a rule, they do not deallocate objects, because convincing a certification authority that dynamic allocation and deallocation of memory is safely used is, in general, quite difficult. This paradigm is diametrically opposed to standard Java, where the design of the language itself requires dynamic memory allocation and garbage collection.

Java augmented by the RTSJ provides three types of memory areas: heap, immortal, and scoped memory. In all types of memory, objects can be explicitly allocated but not explicitly deallocated, thereby ensuring memory consistency. The heap is the standard Java memory area, where a garbage collector is responsible for reclaiming objects that are no longer referenced by the running program. Scoped memory provides region-based memory management similar to allocating objects on a thread's stack and deallocating them when the thread leaves that stack frame. Of the RTSJ memory constructs, only immortal memory is familiar in concept to the

safety-critical software community; objects may be allocated there but never deallocated. Objects may only be reused explicitly by the application.

SCJ does not provide the full spectrum of RTSJ memory areas. Even though there are efficient real-time garbage collectors that might be shown to be certifiable, the jump from the current status quo to such an environment is perceived to be too large for general acceptance, particularly for applications that need to be certified at the highest levels. Likewise, the controversy over the complexity, the expressive power, and the need for runtime checks of the full RTSJ scoped memory model, along with the required programming paradigm shift again suggests that such a "leap of faith" is also beyond current safety-critical software practice.

SCJ provides only immortal memory and limited forms of scoped memory. These limited forms of scoped memory are optimized for a conservative memory model more familiar to safety-critical programmers. The resulting memory model is much simpler than that of the RTSJ.

The outermost memory in an SCJ system is immortal memory. Class objects are allocated in immortal memory as defined in the RTSJ (for further information, see Chapter 3). An SCJ application starts up in immortal memory and can use it to allocate objects that are to be preserved through all missions. For instance, the initial mission sequencer is allocated in immortal memory.

When missions are run, the mission seqeuncer enters a fresh mission memory where the application initializes its mission before its execution phase. This mission memory holds the mission's schedulable objects and any objects shared among these schedulable objects.

Following mission initialization, during execution and clean up, a schedulable object allocates objects in private memories that have been cleared before the schedulable object is released. A schedulable object may enter a sequence of nested private memories during a release. The private memories will be cleared whenever a handler exits its handler logic and when a thread exits its run method. Thus objects that need to survive between releases must be in an outer memory, typically the mission memory. At the end of a mission, when control has returned to the mission sequencer, mission memory will be cleared before another mission is started.

7.5.1 Nesting Scopes

mission memory is just a ScopedMemory which is provided for the application during start up for holding objects that have a mission life span. This acts like an immortal memory area during a mission, except that it will be reinitialized at the end of each mission. All objects needed during a mission for a longer duration than one schedulable object release are allocated in the mission memory area. The mission memory area is exited only after all schedulable objects have terminated.

Because the mission memory is not cleared during the mission, allocation of objects in the mission memory during the execution of the mission can lead to a memory leak; therefore, each schedulable object is given its own private scoped memory, and each schedulable object may create and use additional private memories. Thus, each instance of the event handler classes available to the application programmer has its own PrivateMemory that is entered on each release and exited at the end of each release.

7.5.2 Finalizers

The RTSJ provides for calling finalizers when the last thread exits a scoped memory. Because finalization can cause unpredictable delay, finalizers are not allowed in SCJ.

7.6 Compatibility

In general, the SCJ conforms to the Java memory model. With respect to this memory model, AsynchronousEventHandlers behave like Java threads. Fields accessed from more than one AsynchronousEventHandler should be synchronized or declared volatile to ensure that changes made in the context of one handler are visible in all other handlers which reference the field. Although at Level 0 all AsynchronousEventHandlers are run in single thread context, synchronization should still be specified to aid application portability to other implementation.

SCJ provides its own classes for managing memory. From a programming view, they are compatible with the RTSJ, although some of the management methods are different. Therefore code that uses the SCJ classes would need these classes to run in an RTSJ environment.

Chapter 8

Clocks, Timers, and Time

8.1 Overview

Most safety-critical applications require precise timing mechanisms for maintaining real-time response. SCJ provides a restricted subset of the timing mechanisms of the RTSJ.

The Overview and Rationale sections are not normative but are provided to improve understanding of the normative sections. All of the other sections of this chapter are normative.

8.2 Semantics and Requirements

The semantics the SCJ Time, Clock, and Chronograph classes comprise a subset of the corresponding classes in the RTSJ. Provision is made for applications to read the time from a hardware clock, create events based on time, and to provide application-defined clocks that offer a flexible mechanism to deal with recurring events.

The resolution and drift of any clock, including the default real-time clock, is dependent on the underlying hardware clock and the operating system implementation, if one is present. Application developers should refer to the hardware clock specification as well as information from the OS as well as the SCJ vendor. See Section 4.8.4 for a discussion of the effects of clock granularity.

The resolution returned by a clock's getDrivePrecision() method is the resolution that shall be used for all scheduling decisions based on that clock.

8.2.1 Chronographs and Clocks

A chronograph measures time, whereas a clock is a chronograph that reacts to the passage of time. SCJ shall support a single system real-time clock and a set of application-defined clocks. As in the RTSJ, the real-time clock shall be *weakly increasing* and *monotonic*. The real-time clock in the RTSJ has an Epoch of January 1, 1970. While the *Epoch* for the real-time clock in an SCJ system is generally the same, the *Epoch* in a SCJ system may alternatively represent the system start time if the underlying operating system lacks a way to reliably determine the current date. As a consequence, absolute times based on the real-time clock may not correspond to the wall-clock time.

8.2.2 Time

Two time classes from the RTSJ are available for use in safety critical applications: AbsoluteTime and RelativeTime. As in the RTSJ, the base abstract time class for both of these classes is HighResolutionTime. AbsoluteTime represents a specific point in time, while RelativeTime represents a time interval.

Instances of HighResolutionTime classes always hold a normalized form of a time value. Values that cannot be normalized are not valid; for example, (MAX_LONG milliseconds, MAX_INT nanoseconds) cannot be normalized and therefore represents an illegal value. For additional details of time normalization, see the chapter covering Time in the current RTSJ specification.

8.2.3 Application-defined Chronographs and Clocks

While every SCJ implementation shall provide a default real-time clock, SCJ implementations shall also permit application developers to define application-defined chronographs and clocks. An application-defined clock shall drive events and can therefore be used for scheduling. A application-defined chronograph shall not drive events and shall therefore be read-only. In exactly the same way as the real-time clock, application-defined clocks and chronographs can be referenced in the constructors of objects based on the AbsoluteTime and RelativeTime classes. Application-defined clocks (and consequently infrastructure-defined timers that are based on those clocks) facilitate the release of periodic schedulable objects and timeouts based on application-detected events.

An application-defined clock shall be responsible only for providing the current time and signalling a timing event also called an alarm when a single absolute time has been reached. If an event is requested by the application that is earlier than the current one, the infrastructure informs the application-defined clock to reset its target

time. Any queue management associated with timer functions shall be supported by the SCJ infrastructure. This means that the SCJ infrastructure shall behave as if it maintains a time-ordered list of timing events required by the application with respect to each application-defined clock.

Figure 8.1 illustrates the interactions between the SCJ infrastructure and an application-defined clock. The example starts with a call to the `delay` method in the `Services` class:

1. The application creates an application-defined clock.
2. The application creates a relative time object based on that clock.
3. The application calls `Services.delay` with the relative time value.
4. `Services.delay` calls the infrastructure to request that the current managed schedulable be suspended until the delay time has expired.
5. The SCJ infrastructure determines whether the required timing event is the earliest event associated with the application-defined clock. If it is the earliest event, it calls the `setAlarm` method in the clock. If the required timing event is at or before the current time of the clock, the `delay` function returns. If the required timing event is not the earliest, the managed schedulable that called `delay` is suspended until the earliest event has been handled; then this step is repeated.
6. The application-defined clock performs its application-dependent functionality.
7. When the time has expired, the application-defined clock calls `triggerAlarm`.
8. The SCJ infrastructure reschedules the suspended managed schedulable.
9. The managed schedulable returns from the infrastructure call that caused it to be suspended.
10. The call to `delay` returns.

As required for the default SCJ real-time clock, an application-defined clock should be *weakly increasing* and *monotonic*. The behavior of the infrastructure may be compromised if it is not. As the application-defined clock is driven by application-generated events, the notions of clock *resolution* and *uniformity* shall have an application-defined meaning.

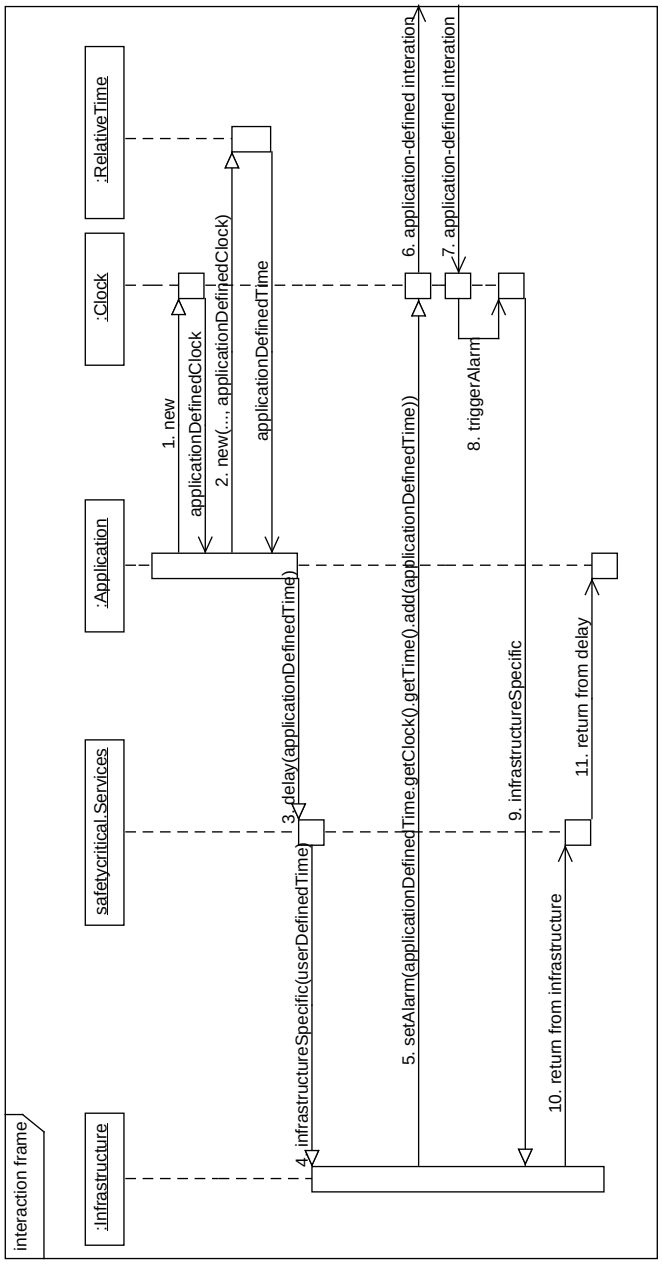

Figure 8.1: Sequence diagram of `delay` using an application-defined clock

8.2.4 RTSJ Constraints

The RTSJ classes OneShotTimer, PeriodicTimer, and Timer that can be used to schedule application logic in the RTSJ are not directly available in SCJ. Periodic, time-triggered application logic is constructed using the PeriodicEventHandler class. For timeouts and non-periodic time-triggered releases of a handler, SCJ provides the OneShotEventHandler class.

The RTSJ provides an interface that allows the program to set the priority and affinity of the infrastructure process that releases handlers resulting from time-triggered events. The SCJ does not support this interface. All SCJ infrastructure code that interacts with clocks is run at the highest priority and with an implementation-defined affinity.

8.3 Level Considerations

Because wait and notify are available only at compliance Level 2, the method waitForObject in HighResolutionTime is available only at compliance Level 2.

Application-defined clocks are available only at Level 1 and Level 2, and are not available at Level 0.

8.4 API

Figure 8.2 gives an overview of the time related classes.

Unless indicated otherwise, the classes defined in this section are thread safe

8.4.1 Class javax.realtime.Chronograph

Declaration

```
@SCJAllowed
public interface Chronograph
```

Description

The interface for all devices that support the measurement of time. All Chronograph implementations use time values derived from HighResolutionTime, which expresses its time in milliseconds and nanoseconds. However, for an application-defined clock, its time values are not necessarily related to the wall clock time in any particular fashion. For instance, they could represent a count of wheel

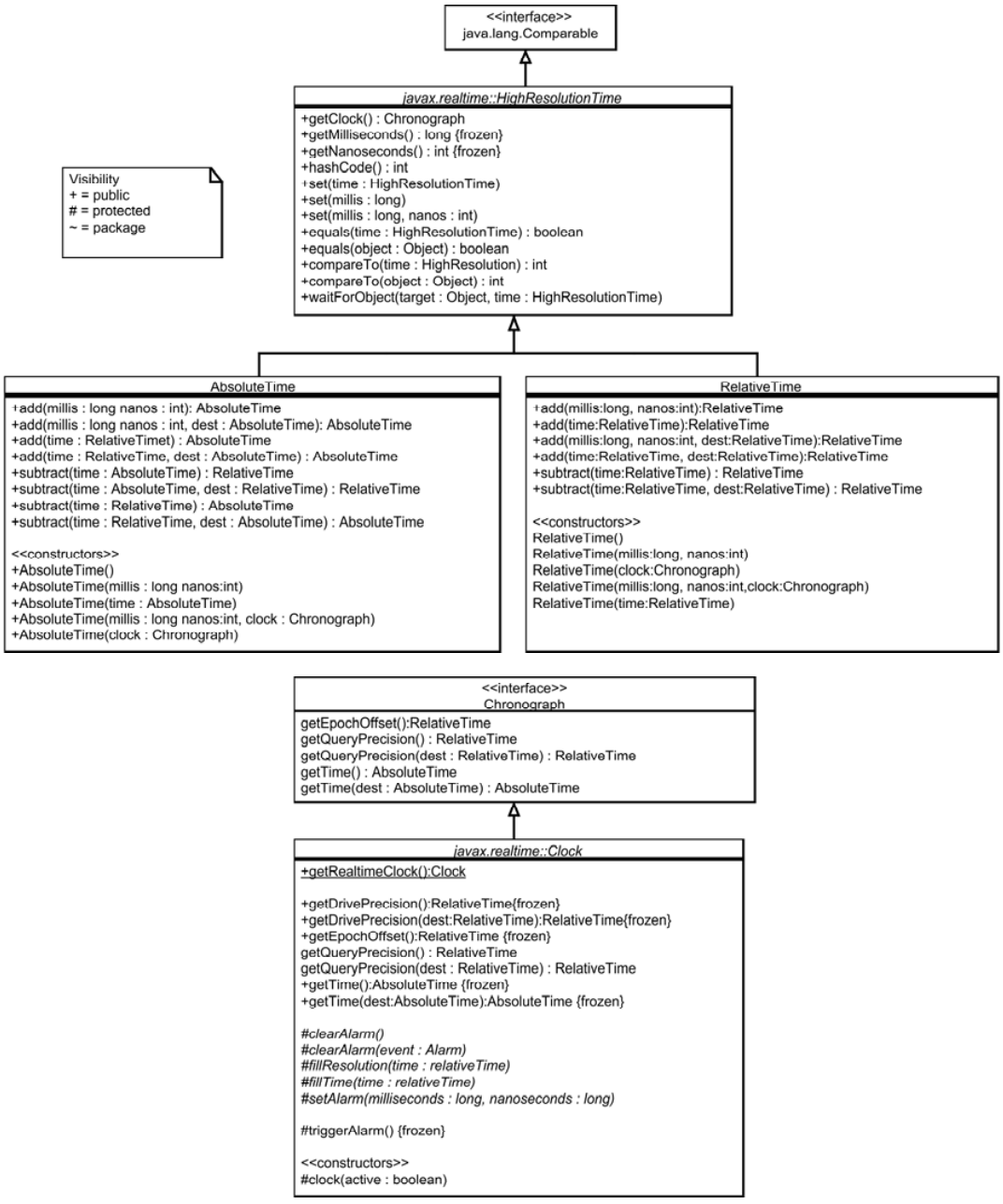

Figure 8.2: Time classes

revolutions or particular event detections. In any case, the time values for every clock shall be mapped to milliseconds and nanoseconds in a manner that is computationally appropriate.

Methods

```
@SCJAllowed
@SCJMaySelfSuspend(false)
@SCJMayAllocate({javax.safetycritical.annotate.AllocationContext.CURRENT})
@SCJPhase({
    javax.safetycritical.annotate.Phase.STARTUP,
    javax.safetycritical.annotate.Phase.INITIALIZATION,
    javax.safetycritical.annotate.Phase.RUN,
    javax.safetycritical.annotate.Phase.CLEANUP })
public javax.realtime.RelativeTime getEpochOffset( )
```

Determines the time on the real-time clock when this chronograph was zero.

returns A newly allocated RelativeTime object with the real-time clock as its chronograph and containing the time from the real-time clock when this chronograph was zero.

Throws UnsupportedOperationException when this chronograph does not have the concept of an Epoch.

```
@SCJAllowed
@SCJMaySelfSuspend(false)
@SCJMayAllocate({javax.safetycritical.annotate.AllocationContext.CURRENT})
@SCJPhase({
    javax.safetycritical.annotate.Phase.STARTUP,
    javax.safetycritical.annotate.Phase.INITIALIZATION,
    javax.safetycritical.annotate.Phase.RUN,
    javax.safetycritical.annotate.Phase.CLEANUP })
public javax.realtime.RelativeTime getQueryPrecision(RelativeTime dest)
```

Gets the precision of the time read, defined as the nominal interval between ticks.

dest — is an object that, upon return from this method, shall contain the precision of the time read. If dest is null, this method shall allocate a new RelativeTime instance to hold the returned value.

returns the value of dest if dest is not null, otherwise a new object representing the read precision.

```
@SCJAllowed
@SCJMaySelfSuspend(false)
@SCJMayAllocate({javax.safetycritical.annotate.AllocationContext.CURRENT})
@SCJPhase({
   javax.safetycritical.annotate.Phase.STARTUP,
   javax.safetycritical.annotate.Phase.INITIALIZATION,
   javax.safetycritical.annotate.Phase.RUN,
   javax.safetycritical.annotate.Phase.CLEANUP })
public javax.realtime.RelativeTime getQueryPrecision( )
```

Gets the precision of the time read defined as the nominal interval between ticks. It is the same as calling getQueryPrecision(null).

returns a value representing the read precision.

```
@SCJAllowed
@SCJMaySelfSuspend(false)
@SCJMayAllocate({})
@SCJPhase({
   javax.safetycritical.annotate.Phase.STARTUP,
   javax.safetycritical.annotate.Phase.INITIALIZATION,
   javax.safetycritical.annotate.Phase.RUN,
   javax.safetycritical.annotate.Phase.CLEANUP })
public javax.realtime.AbsoluteTime getTime(AbsoluteTime dest)
```

Gets the current time of this chronograph. The time represented by the returned AbsoluteTime represents some time between the invocation of the method and the return of the method. *Note:* This method will return an absolute time value that represents the chronograph's notion of the current time. For chronographs that do not measure calendar time this absolute time may not represent a wall clock time.

dest — The instance of an AbsoluteTime object which will be updated in place. When dest is not null, the clock association of the dest parameter at the time of the call is ignored; the returned object will be associated with this chronograph. When dest is null, nothing happens.

returns the instance of AbsoluteTime passed as a parameter, representing the current time, associated with this chronograph, or null when dest is null.

```
@SCJAllowed
@SCJMaySelfSuspend(false)
@SCJMayAllocate({javax.safetycritical.annotate.AllocationContext.CURRENT})
@SCJPhase({
   javax.safetycritical.annotate.Phase.STARTUP,
   javax.safetycritical.annotate.Phase.INITIALIZATION,
```

```
    javax.safetycritical.annotate.Phase.RUN,
    javax.safetycritical.annotate.Phase.CLEANUP })
public javax.realtime.AbsoluteTime getTime( )
```

Gets the current time in a newly allocated object. The time represented by the returned AbsoluteTime represents some time between the invocation of the method and the return of the method. *Note:* This method will return an absolute time value that represents the chronograph's notion of the current time. For chronographs that do not measure calendar time this absolute time may not represent a wall clock time.

returns a newly allocated instance of AbsoluteTime in the current allocation context, representing the current time. The returned object is associated with this chronograph.

8.4.2 Class javax.realtime.Clock

Declaration

```
@SCJAllowed
 public abstract class Clock implements javax.realtime.Chronograph extends
     java.lang.Object
```

Description

A clock is a chronograph that also manages time events (also called alarms) that can be queued on it and that will cause an event handler to be released when their appointed time is reached.

The Clock instance returned by getRealtimeClock may be used in any context that requires a clock.

HighResolutionTime instances that use application-defined clocks are valid for all APIs in SCJ that take HighResolutionTime time types as parameters.

Constructors

```
@SCJMayAllocate({})
@SCJAllowed(javax.safetycritical.annotate.Level.LEVEL_1)
@SCJMaySelfSuspend(false)
@SCJPhase({
    javax.safetycritical.annotate.Phase.STARTUP,
    javax.safetycritical.annotate.Phase.INITIALIZATION,
    javax.safetycritical.annotate.Phase.RUN,
    javax.safetycritical.annotate.Phase.CLEANUP })
public Clock( )
```

Constructor for the abstract class.

Methods

```
@SCJAllowed(javax.safetycritical.annotate.Level.LEVEL_1)
@SCJMayAllocate({})
@SCJMaySelfSuspend(false)
@SCJPhase({
    javax.safetycritical.annotate.Phase.STARTUP,
    javax.safetycritical.annotate.Phase.INITIALIZATION,
    javax.safetycritical.annotate.Phase.RUN,
    javax.safetycritical.annotate.Phase.CLEANUP })
protected abstract void clearAlarm( )
```

Implemented by subclasses to cancel the current outstanding alarm.

```
@SCJMayAllocate({javax.safetycritical.annotate.AllocationContext.CURRENT})
@SCJAllowed
@SCJMaySelfSuspend(false)
@SCJPhase({
    javax.safetycritical.annotate.Phase.STARTUP,
    javax.safetycritical.annotate.Phase.INITIALIZATION,
    javax.safetycritical.annotate.Phase.RUN,
    javax.safetycritical.annotate.Phase.CLEANUP })
public abstract javax.realtime.RelativeTime getDrivePrecision( )
```

Gets the precision of the clock for driving events, It is the same as calling getDrivePrecision(null) .

returns a value representing the drive precision.

```
@SCJMayAllocate({javax.safetycritical.annotate.AllocationContext.CURRENT})
@SCJAllowed
@SCJMaySelfSuspend(false)
@SCJPhase({
    javax.safetycritical.annotate.Phase.STARTUP,
    javax.safetycritical.annotate.Phase.INITIALIZATION,
    javax.safetycritical.annotate.Phase.RUN,
    javax.safetycritical.annotate.Phase.CLEANUP })
public abstract javax.realtime.RelativeTime getDrivePrecision(RelativeTime dest)
```

Gets the precision of the clock for driving events, defined as the nominal interval between ticks that can trigger an event. This is the resolution that shall be used for all scheduling decisions based on this clock. The result may be larger than that of getQueryPrecision() .

dest — returns the precision in dest. When dest is null, it allocates a new RelativeTime instance to hold the returned value.

returns the value of dest if dest is not null, otherwise a new object representing the drive precision.

```
@SCJMayAllocate({javax.safetycritical.annotate.AllocationContext.CURRENT})
@SCJAllowed(javax.safetycritical.annotate.Level.LEVEL_1)
@SCJMaySelfSuspend(false)
@SCJPhase({
   javax.safetycritical.annotate.Phase.STARTUP,
   javax.safetycritical.annotate.Phase.INITIALIZATION,
   javax.safetycritical.annotate.Phase.RUN,
   javax.safetycritical.annotate.Phase.CLEANUP })
public final javax.realtime.RelativeTime getEpochOffset( )
```

Determines the time on the real-time clock when this clcok was zero.

returns A newly allocated RelativeTime object in the current execution context with the real-time clock as its chronograph and containing the time when this chronograph was zero.

Throws UnsupportedOperationException when the clock does not have the concept of Epoch.

```
@SCJMayAllocate({javax.safetycritical.annotate.AllocationContext.CURRENT})
@SCJAllowed
@SCJMaySelfSuspend(false)
@SCJPhase({
   javax.safetycritical.annotate.Phase.STARTUP,
   javax.safetycritical.annotate.Phase.INITIALIZATION,
   javax.safetycritical.annotate.Phase.RUN,
   javax.safetycritical.annotate.Phase.CLEANUP })
public abstract javax.realtime.RelativeTime getQueryPrecision( )
```

Gets the precision of the time read, defined as the nominal interval between ticks. It is the same as calling getQueryPrecision(null) .

returns the value representing the read precision.

```
@SCJMayAllocate({javax.safetycritical.annotate.AllocationContext.CURRENT})
@SCJAllowed
@SCJMaySelfSuspend(false)
@SCJPhase({
   javax.safetycritical.annotate.Phase.STARTUP,
   javax.safetycritical.annotate.Phase.INITIALIZATION,
   javax.safetycritical.annotate.Phase.RUN,
   javax.safetycritical.annotate.Phase.CLEANUP })
public abstract javax.realtime.RelativeTime getQueryPrecision(RelativeTime dest)
```

Gets the precision of the time read, defined as the nominal interval between ticks. The result may be smaller than that of getDrivePrecision(), when the clock is tied to some system tick for releasing time events.

dest — returns the relative time value in dest. When dest is null, allocate a new RelativeTime instance to hold the returned value.

returns the value of dest if dest is not null, otherwise a new object representing the read precision.

```
@SCJAllowed
@SCJMaySelfSuspend(false)
@SCJMayAllocate({})
@SCJPhase({
   javax.safetycritical.annotate.Phase.STARTUP,
   javax.safetycritical.annotate.Phase.INITIALIZATION,
   javax.safetycritical.annotate.Phase.RUN,
   javax.safetycritical.annotate.Phase.CLEANUP })
public static javax.realtime.Clock getRealtimeClock( )
```

There is always at least one clock object available: the system real-time clock. This is the default Clock.

returns the singleton instance of the default Clock.

```
@SCJAllowed
@SCJMayAllocate({})
@SCJMaySelfSuspend(false)
@SCJPhase({
   javax.safetycritical.annotate.Phase.STARTUP,
   javax.safetycritical.annotate.Phase.INITIALIZATION,
   javax.safetycritical.annotate.Phase.RUN,
   javax.safetycritical.annotate.Phase.CLEANUP })
public abstract javax.realtime.AbsoluteTime getTime(AbsoluteTime dest)
```

Gets the current time in an existing object. The time represented by the given AbsoluteTime is changed at some time between the invocation of the method and the return of the method. This method will return an absolute time value that represents the clock's notion of the current absolute time. For clocks that do not measure calendar time, this absolute time may not represent a wall clock time.

dest — The instance of AbsoluteTime object that will be updated in place. The clock association of the dest parameter is overwritten. When dest is not null the returned object is associated with this clock. If dest is null, then nothing happens.

returns the instance of AbsoluteTime passed as a parameter, representing the current time, associated with this clock, or null if dest was null.

```
@SCJMayAllocate({javax.safetycritical.annotate.AllocationContext.CURRENT})
@SCJAllowed
@SCJMaySelfSuspend(false)
@SCJPhase({
   javax.safetycritical.annotate.Phase.STARTUP,
   javax.safetycritical.annotate.Phase.INITIALIZATION,
   javax.safetycritical.annotate.Phase.RUN,
   javax.safetycritical.annotate.Phase.CLEANUP })
public final javax.realtime.AbsoluteTime getTime( )
```

Gets the current time in a newly allocated object. This method will return an absolute time value that represents the clock's notion of an absolute time. For clocks that do not measure calendar time, this absolute time may not represent a wall clock time.

returns a newly allocated instance of AbsoluteTime in the current allocation context, representing the current time. The returned object is associated with this clock.

```
@SCJAllowed(javax.safetycritical.annotate.Level.LEVEL_1)
@SCJMayAllocate({})
@SCJMaySelfSuspend(false)
@SCJPhase({
   javax.safetycritical.annotate.Phase.STARTUP,
   javax.safetycritical.annotate.Phase.INITIALIZATION,
   javax.safetycritical.annotate.Phase.RUN,
   javax.safetycritical.annotate.Phase.CLEANUP })
protected abstract void setAlarm(long milliseconds, int nanoseconds)
```

Implemented by subclasses to set the time for the next alarm. If there is an alarm outstanding when called, it overwrites the old time. The milliseconds and nanoseconds are interpreted as an absolute time.

milliseconds — of the next alarm.

nanoseconds — of the next alarm.

```
@SCJAllowed(javax.safetycritical.annotate.Level.LEVEL_1)
@SCJMayAllocate({})
@SCJMaySelfSuspend(false)
@SCJPhase({
   javax.safetycritical.annotate.Phase.STARTUP,
   javax.safetycritical.annotate.Phase.INITIALIZATION,
   javax.safetycritical.annotate.Phase.RUN,
   javax.safetycritical.annotate.Phase.CLEANUP })
protected final void triggerAlarm( )
```

Called by a subclass to signal that the time of the next alarm has been reached.

8.4.3 Class javax.realtime.HighResolutionTime

Declaration

```
@SCJAllowed
 public abstract class HighResolutionTime<T extends HighResolutionTime<T>>
     implements java.lang.Comparable<T>, java.lang.Cloneable extends
     java.lang.Object
```

Description

Class HighResolutionTime is the abstract base class for AbsoluteTime and RelativeTime, and is used to express time with nanosecond accuracy. When an API is defined that has an HighResolutionTime as a parameter, it can take either an absolute or relative time and will do something appropriate.

A time object in normalized form represents negative time if both components are nonzero and negative, or one is nonzero and negative and the other is zero. For add and subtract negative values behave as they do in arithmetic.

Caution: This class is explicitly unsafe in multithreaded situations when it is being changed. No synchronization is done. It is assumed that users of this class who are mutating instances will be doing their own synchronization at a higher level.

Methods

```
@SCJAllowed
@SCJMaySelfSuspend(false)
@SCJMayAllocate({})
@SCJPhase({
   javax.safetycritical.annotate.Phase.STARTUP,
   javax.safetycritical.annotate.Phase.INITIALIZATION,
   javax.safetycritical.annotate.Phase.RUN,
   javax.safetycritical.annotate.Phase.CLEANUP })
public int compareTo(T time)
```

Compares thisHighResolutionTime with the specified HighResolutionTime time.

time — Compares with the time of this.

Throws ClassCastException if the time parameter is not of the same class as this.

Throws IllegalArgumentException if the time parameter is not associated with the same clock as this, or when the time parameter is null.

returns a negative integer, zero, or a positive integer as this object is less than, equal to, or greater than time.

```
@SCJAllowed
@SCJMaySelfSuspend(false)
@SCJMayAllocate({})
@SCJPhase({
    javax.safetycritical.annotate.Phase.STARTUP,
    javax.safetycritical.annotate.Phase.INITIALIZATION,
    javax.safetycritical.annotate.Phase.RUN,
    javax.safetycritical.annotate.Phase.CLEANUP })
public boolean equals(T time)
```

Returns true if the argument time has the same type and values as this.

Equality includes clock association.

time — Value compared to this.

returns true if the parameter time is of the same type and has the same values as this.

```
@SCJAllowed
@SCJMaySelfSuspend(false)
@SCJMayAllocate({})
@SCJPhase({
    javax.safetycritical.annotate.Phase.STARTUP,
    javax.safetycritical.annotate.Phase.INITIALIZATION,
    javax.safetycritical.annotate.Phase.RUN,
    javax.safetycritical.annotate.Phase.CLEANUP })
public boolean equals(Object object)
```

Returns true if the argument object has the same type and values as this.

Equality includes clock association.

object — Value compared to this.

returns true if the parameter object is of the same type and has the same values as this.

```
@SCJAllowed
@SCJMaySelfSuspend(false)
@SCJMayAllocate({})
@SCJPhase({
    javax.safetycritical.annotate.Phase.STARTUP,
    javax.safetycritical.annotate.Phase.INITIALIZATION,
    javax.safetycritical.annotate.Phase.RUN,
    javax.safetycritical.annotate.Phase.CLEANUP })
public javax.realtime.Chronograph getChronograph( )
```

Get a reference to the **javax.realtime.Chronograph** associated with this.

returns a reference to the **javax.realtime.Chronograph** associated with this.

```
@SCJAllowed
@SCJMaySelfSuspend(false)
@SCJMayAllocate({})
@SCJPhase({
    javax.safetycritical.annotate.Phase.STARTUP,
    javax.safetycritical.annotate.Phase.INITIALIZATION,
    javax.safetycritical.annotate.Phase.RUN,
    javax.safetycritical.annotate.Phase.CLEANUP })
public javax.realtime.Clock getClock( )
```

Returns a reference to the clock associated with this.

returns a reference to the clock associated with this.

Throws UnsupportedOperationException if the time is based on a **javax.realtime-.Chronograph** that is not a **javax.realtime.Clock** .

```
@SCJAllowed
@SCJMaySelfSuspend(false)
@SCJMayAllocate({})
@SCJPhase({
    javax.safetycritical.annotate.Phase.STARTUP,
    javax.safetycritical.annotate.Phase.INITIALIZATION,
    javax.safetycritical.annotate.Phase.RUN,
    javax.safetycritical.annotate.Phase.CLEANUP })
public final long getMilliseconds( )
```

Returns the milliseconds component of this.

returns the milliseconds component of the time represented by this.

```
@SCJAllowed
@SCJMaySelfSuspend(false)
@SCJMayAllocate({})
@SCJPhase({
    javax.safetycritical.annotate.Phase.STARTUP,
    javax.safetycritical.annotate.Phase.INITIALIZATION,
    javax.safetycritical.annotate.Phase.RUN,
    javax.safetycritical.annotate.Phase.CLEANUP })
public final int getNanoseconds( )
```

Returns the nanoseconds component of this.

returns the nanoseconds component of the time represented by this.

```
@SCJAllowed
@SCJMaySelfSuspend(false)
@SCJMayAllocate({})
@SCJPhase({
   javax.safetycritical.annotate.Phase.STARTUP,
   javax.safetycritical.annotate.Phase.INITIALIZATION,
   javax.safetycritical.annotate.Phase.RUN,
   javax.safetycritical.annotate.Phase.CLEANUP })
public int hashCode( )
```

Returns a hash code for this object in accordance with the general contract of hashCode. Time objects that are equals(HighResolutionTime) have the same hash code.

returns the hashcode value for this instance.

```
@SCJAllowed
@SCJMaySelfSuspend(false)
@SCJMayAllocate({})
@SCJPhase({
   javax.safetycritical.annotate.Phase.STARTUP,
   javax.safetycritical.annotate.Phase.INITIALIZATION,
   javax.safetycritical.annotate.Phase.RUN,
   javax.safetycritical.annotate.Phase.CLEANUP })
public T set(long millis)
```

Sets the millisecond component of this to the given argument, and the nanosecond component of this to 0. This method is equivalent to set(millis, 0).

millis — The desired value of the millisecond component of this.

```
@SCJAllowed
@SCJMaySelfSuspend(false)
@SCJMayAllocate({})
@SCJPhase({
   javax.safetycritical.annotate.Phase.STARTUP,
   javax.safetycritical.annotate.Phase.INITIALIZATION,
   javax.safetycritical.annotate.Phase.RUN,
   javax.safetycritical.annotate.Phase.CLEANUP })
public T set(long millis, int nanos)
```

Sets the millisecond and nanosecond components of this. The setting is subject to parameter normalization. If there is an overflow in the millisecond component while normalizing then an IllegalArgumentException will be thrown.

millis — The desired value for the millisecond component of this before normalization.

nanos — The desired value for the nanosecond component of this before normalization.

Throws IllegalArgumentException if there is an overflow in the millisecond component while normalizing.

```
@SCJAllowed
@SCJMaySelfSuspend(false)
@SCJMayAllocate({})
@SCJPhase({
   javax.safetycritical.annotate.Phase.STARTUP,
   javax.safetycritical.annotate.Phase.INITIALIZATION,
   javax.safetycritical.annotate.Phase.RUN,
   javax.safetycritical.annotate.Phase.CLEANUP })
public T set(T time)
```

Change the value represented by this to that of the given time. If the time parameter is null this method will throw IllegalArgumentException. If the type of this and the type of the given time are not the same, this method will throw ClassCastException. The clock associated with this is set to be the clock associated with the time parameter.

time — The new value for this.

Throws IllegalArgumentException if the parameter time is null.

Throws ClassCastException if the type of this and the type of the parameter time are not the same.

```
@SCJAllowed(javax.safetycritical.annotate.Level.LEVEL_2)
@SCJMayAllocate({})
@SCJMaySelfSuspend(true)
@SCJPhase({
   javax.safetycritical.annotate.Phase.STARTUP,
   javax.safetycritical.annotate.Phase.INITIALIZATION,
   javax.safetycritical.annotate.Phase.RUN,
   javax.safetycritical.annotate.Phase.CLEANUP })
public static boolean waitForObject(Object target, HighResolutionTime<?> time)
  throws java.lang.InterruptedException
```

Behaves exactly like target.wait() but with the enhancement that it waits with a precision of HighResolutionTime.

The wait time may be relative or absolute, and it is controlled by the clock associated with it. If the wait time is relative, then the calling thread is blocked waiting on target for the amount of time given by time, and measured by the associated clock. If the wait time is absolute, then the calling thread is blocked waiting on target until the indicated time value is reached by the associated clock.

target — The object on which to wait. The current thread must have a lock on the object.

time — The time for which to wait. If it is RelativeTime(0,0) then wait indefinitely. If it is null then wait indefinitely.

returns true if a notify was received before the timeout, False otherwise.

Throws InterruptedException if this schedulable object is interrupted by Realtime-Thread.interrupt.

Throws IllegalArgumentException if time represents a relative time less than zero.

Throws IllegalMonitorStateException if target is not locked by the caller.

Throws UnsupportedOperationException if the wait operation is not supported using the clock associated with time.

See Also: java.lang.Object.wait(), java.lang.Object.wait(long), java.lang.Object.wait(long,int)

8.4.4 Class javax.realtime.AbsoluteTime

Declaration

```
@SCJAllowed
 public class AbsoluteTime extends javax.realtime.HighResolutionTime
```

Description

An object that represents a specific point in time given by milliseconds plus nanoseconds past some point in time fixed by its associated clock. For the default real-time clock the fixed point is the implementation dependent Epoch.

The correctness of the Epoch as a time base depends on the real-time clock synchronization with an external world time reference. This representation was designed to be compatible with the standard Java representation of an absolute time in the java.util.Date class.

Caution: This class is explicitly unsafe in multithreaded situations when it is being changed. No synchronization is done. It is assumed that users of this class who are mutating instances will be doing their own synchronization at a higher level.

Constructors

```
@SCJAllowed
@SCJPhase({
   javax.safetycritical.annotate.Phase.INITIALIZATION,
   javax.safetycritical.annotate.Phase.RUN,
   javax.safetycritical.annotate.Phase.CLEANUP})
```

```
@SCJMaySelfSuspend(false)
@SCJMayAllocate({})
public AbsoluteTime(long millis, int nanos, Chronograph clock)
```

Construct an AbsoluteTime object with time millisecond and nanosecond components past the Epoch for clock.

The value of the AbsoluteTime instance is based on the parameter millis plus the parameter nanos. The construction is subject to millis and nanos parameters normalization. If there is an overflow in the millisecond component when normalizing then an IllegalArgumentException will be thrown. If after normalization the time object is negative then the time represented by this is time before the Epoch.

The clock association is made with the clock parameter.

This constructor requires that the ”clock” parameter resides in a scope that encloses the scope of the ”this” argument.

millis — The desired value for the millisecond component of this. The actual value is the result of parameter normalization.

nanos — The desired value for the nanosecond component of this. The actual value is the result of parameter normalization.

clock — The chronograph providing the association for the newly constructed object. If clock is null the association is made with the real-time clock.

Throws IllegalArgumentException if there is an overflow in the millisecond component when normalizing.

```
@SCJAllowed
@SCJPhase({
   javax.safetycritical.annotate.Phase.INITIALIZATION,
   javax.safetycritical.annotate.Phase.RUN,
   javax.safetycritical.annotate.Phase.CLEANUP})
@SCJMaySelfSuspend(false)
@SCJMayAllocate({})
public AbsoluteTime(long millis, int nanos)
```

This constructor behaves the same as calling AbsoluteTime(millis, nanos, null).

```
@SCJAllowed
@SCJPhase({
   javax.safetycritical.annotate.Phase.INITIALIZATION,
   javax.safetycritical.annotate.Phase.RUN,
   javax.safetycritical.annotate.Phase.CLEANUP})
@SCJMaySelfSuspend(false)
@SCJMayAllocate({})
public AbsoluteTime( )
```

This constructor behaves the same as calling AbsoluteTime(0, 0, null).

```
@SCJAllowed
@SCJPhase({
   javax.safetycritical.annotate.Phase.INITIALIZATION,
   javax.safetycritical.annotate.Phase.RUN,
   javax.safetycritical.annotate.Phase.CLEANUP})
@SCJMaySelfSuspend(false)
@SCJMayAllocate({})
public AbsoluteTime(Chronograph clock)
```

This constructor behaves the same as calling AbsoluteTime(0, 0, clock).

```
@SCJAllowed
@SCJPhase({
   javax.safetycritical.annotate.Phase.INITIALIZATION,
   javax.safetycritical.annotate.Phase.RUN,
   javax.safetycritical.annotate.Phase.CLEANUP})
@SCJMaySelfSuspend(false)
@SCJMayAllocate({})
public AbsoluteTime(AbsoluteTime time)
```

Make a new AbsoluteTime object from the given AbsoluteTime object. The new object will have the same clock association as the time parameter.

This constructor requires that the ”time” parameter resides in a scope that encloses the scope of the ”this” argument.

time — The AbsoluteTime object which is the source for the copy.

Throws IllegalArgumentException if the time parameter is null.

Methods

```
@SCJAllowed
@SCJPhase({
   javax.safetycritical.annotate.Phase.INITIALIZATION,
   javax.safetycritical.annotate.Phase.RUN,
   javax.safetycritical.annotate.Phase.CLEANUP})
@SCJMaySelfSuspend(false)
@SCJMayAllocate({javax.safetycritical.annotate.AllocationContext.CURRENT})
public javax.realtime.AbsoluteTime add(long millis, int nanos, AbsoluteTime dest)
```

Return an object containing the value resulting from adding millis and nanos to the values from this and normalizing the result. If dest is not null, the result is placed there and returned. Otherwise, a new object is allocated for the result.

The result will have the same clock association as this, and the clock association of dest is ignored.

An ArithmeticException is thrown if the result does not fit in the normalized format.

millis — The number of milliseconds to be added to this.

nanos — The number of nanoseconds to be added to this.

dest — If dest is not null, the result is placed there. Otherwise, a new object is allocated for the result.

returns the value of dest if dest is not null, otherwise a new object representing the result.

Throws ArithmeticException if the result does not fit in the normalized format.

```
@SCJAllowed
@SCJPhase({
    javax.safetycritical.annotate.Phase.INITIALIZATION,
    javax.safetycritical.annotate.Phase.RUN,
    javax.safetycritical.annotate.Phase.CLEANUP})
@SCJMaySelfSuspend(false)
@SCJMayAllocate({javax.safetycritical.annotate.AllocationContext.CURRENT})
public javax.realtime.AbsoluteTime add(RelativeTime time, AbsoluteTime dest)
```

Return an object containing the value resulting from adding time to the value of this and normalizing the result. If dest is not null, the result is placed there and returned. Otherwise, a new object is allocated for the result.

The clock associated with this and the clock associated with the time parameter must be the same, and such association is used for the result.

The clock associated with the dest parameter is ignored.

An IllegalArgumentException is thrown if the clock associated with this and the clock associated with the time parameter are different.

An IllegalArgumentException is thrown if the time parameter is null.

An ArithmeticException is thrown if the result does not fit in the normalized format.

time — The time to add to this.

dest — If dest is not null, the result is placed there and returned. Otherwise, a new object is allocated for the result.

returns the value of dest if dest is not null, otherwise a new object representing the result.

Throws IllegalArgumentException if the clock associated with this and the clock associated with the time parameter are different, or when the time parameter is null.

Throws ArithmeticException if the result does not fit in the normalized format.

```
@SCJAllowed
@SCJPhase({
   javax.safetycritical.annotate.Phase.INITIALIZATION,
   javax.safetycritical.annotate.Phase.RUN,
   javax.safetycritical.annotate.Phase.CLEANUP})
@SCJMaySelfSuspend(false)
@SCJMayAllocate({javax.safetycritical.annotate.AllocationContext.CURRENT})
public javax.realtime.AbsoluteTime add(RelativeTime time)
```

Create a new instance of AbsoluteTime representing the result of adding time to the value of this and normalizing the result.

The clock associated with this and the clock associated with the time parameter must be the same, and such association is used for the result.

An IllegalArgumentException is thrown if the clock associated with this and the clock associated with the time parameter are different.

An IllegalArgumentException is thrown if the time parameter is null.

An ArithmeticException is thrown if the result does not fit in the normalized format.

time — The time to add to this.

returns A new AbsoluteTime object whose time is the normalization of this plus the parameter time.

Throws IllegalArgumentException if the clock associated with this and the clock associated with the time parameter are different, or when the time parameter is null.

Throws ArithmeticException if the result does not fit in the normalized format.

```
@SCJAllowed
@SCJPhase({
   javax.safetycritical.annotate.Phase.INITIALIZATION,
   javax.safetycritical.annotate.Phase.RUN,
   javax.safetycritical.annotate.Phase.CLEANUP})
@SCJMaySelfSuspend(false)
@SCJMayAllocate({javax.safetycritical.annotate.AllocationContext.CURRENT})
public javax.realtime.AbsoluteTime add(long millis, int nanos)
```

Create a new object representing the result of adding millis and nanos to the values from this and normalizing the result. The result will have the same clock association as this. An ArithmeticException is thrown if the result does not fit in the normalized format.

millis — The number of milliseconds to be added to this.

nanos — The number of nanoseconds to be added to this.

returns A new AbsoluteTime object whose time is the normalization of this plus millis and nanos.

Throws ArithmeticException if the result does not fit in the normalized format.

```
@SCJAllowed
@SCJMaySelfSuspend(false)
@SCJMayAllocate({})
@Override
@SCJPhase({
    javax.safetycritical.annotate.Phase.STARTUP,
    javax.safetycritical.annotate.Phase.INITIALIZATION,
    javax.safetycritical.annotate.Phase.RUN,
    javax.safetycritical.annotate.Phase.CLEANUP })
public int compareTo(AbsoluteTime time)
```

Compares this object with the specified object for order.

returns a negative integer, zero, or a positive integer as this object is less than, equal to, or greater than the specified object.

Throws ClassCastException if the time parameter is not of the same class as this.

Throws IllegalArgumentException if the time parameter is not associated with the same clock as this, or when the time parameter is null.

```
@SCJAllowed
@SCJPhase({
    javax.safetycritical.annotate.Phase.INITIALIZATION,
    javax.safetycritical.annotate.Phase.RUN,
    javax.safetycritical.annotate.Phase.CLEANUP})
@SCJMaySelfSuspend(false)
@SCJMayAllocate({javax.safetycritical.annotate.AllocationContext.CURRENT})
public javax.realtime.RelativeTime subtract(AbsoluteTime time, RelativeTime dest)
```

Return an object containing the value resulting from subtracting time from the value of this and normalizing the result. If dest is not null, the result is placed there and returned. Otherwise, a new object is allocated for the result. The clock associated with this and the clock associated with the time parameter must be the same, and such association is used for the result.

The clock associated with the dest parameter is ignored.

An IllegalArgumentException is thrown if the clock associated with this and the clock associated with the time parameter are different. An IllegalArgument-Exception is thrown if the time parameter is null.

An ArithmeticException is thrown if the result does not fit in the normalized format.

time — The time to subtract from this.

dest — If dest is not null, the result is placed there and returned. Otherwise, a new object is allocated for the result.

returns the value of dest if dest is not null, otherwise a new object representing the result.

Throws IllegalArgumentException if the clock associated with this and the clock associated with the time parameter are different, or when the time parameter is null.

Throws ArithmeticException if the result does not fit in the normalized format.

```
@SCJAllowed
@SCJPhase({
   javax.safetycritical.annotate.Phase.INITIALIZATION,
   javax.safetycritical.annotate.Phase.RUN,
   javax.safetycritical.annotate.Phase.CLEANUP})
@SCJMaySelfSuspend(false)
@SCJMayAllocate({javax.safetycritical.annotate.AllocationContext.CURRENT})
public javax.realtime.AbsoluteTime subtract(RelativeTime time, AbsoluteTime dest)
```

Return an object containing the value resulting from subtracting time from the value of this and normalizing the result.

time — The time to subtract from this.

dest — If dest is not null, the result is placed there and returned. Otherwise, a new object is allocated for the result.

returns the value of dest if dest is not null, otherwise a new object representing the result.

```
@SCJAllowed
@SCJPhase({
   javax.safetycritical.annotate.Phase.INITIALIZATION,
   javax.safetycritical.annotate.Phase.RUN,
   javax.safetycritical.annotate.Phase.CLEANUP})
@SCJMaySelfSuspend(false)
@SCJMayAllocate({javax.safetycritical.annotate.AllocationContext.CURRENT})
public javax.realtime.AbsoluteTime subtract(RelativeTime time)
```

Create a new instance of AbsoluteTime representing the result of subtracting time from the value of this and normalizing the result.

The clock associated with this and the clock associated with the time parameter must be the same, and such association is used for the result.

An IllegalArgumentException is thrown if the clock associated with this and the clock associated with the time parameter are different. An IllegalArgument-Exception is thrown if the time parameter is null.

An ArithmeticException is thrown if the result does not fit in the normalized format.

time — The time to subtract from this.

returns A new AbsoluteTime object whose time is the normalization of this minus the parameter time.

Throws IllegalArgumentException if the clock associated with this and the clock associated with the time parameter are different, or when the time parameter is null.

Throws ArithmeticException if the result does not fit in the normalized format.

```
@SCJAllowed
@SCJPhase({
    javax.safetycritical.annotate.Phase.INITIALIZATION,
    javax.safetycritical.annotate.Phase.RUN,
    javax.safetycritical.annotate.Phase.CLEANUP})
@SCJMaySelfSuspend(false)
@SCJMayAllocate({javax.safetycritical.annotate.AllocationContext.CURRENT})
public javax.realtime.RelativeTime subtract(AbsoluteTime time)
```

Create a new instance of RelativeTime representing the result of subtracting time from the value of this and normalizing the result.

The clock associated with this and the clock associated with the time parameter must be the same, and such association is used for the result.

An IllegalArgumentException is thrown if the clock associated with this and the clock associated with the time parameter are different.

An IllegalArgumentException is thrown if the time parameter is null.

An ArithmeticException is thrown if the result does not fit in the normalized format.

time — The time to subtract from this.

returns A new RelativeTime object whose time is the normalization of this minus the AbsoluteTime parameter time.

Throws IllegalArgumentException if the clock associated with this and the clock associated with the time parameter are different, or when the time parameter is null.

Throws ArithmeticException if the result does not fit in the normalized format.

8.4.5 Class javax.realtime.RelativeTime

Declaration

```
@SCJAllowed
 public class RelativeTime extends javax.realtime.HighResolutionTime
```

Description

An object that represents a time interval represented by a number of milliseconds plus nanoseconds. The time interval is kept in normalized form.

A negative interval relative to now represents time in the past. For add and subtract negative values behave as they do in arithmetic.

Caution: This class is explicitly unsafe in multithreaded situations when it is being changed. No synchronization is done. It is assumed that users of this class who are mutating instances will be doing their own synchronization at a higher level.

Constructors

```
@SCJAllowed
@SCJMaySelfSuspend(false)
@SCJMayAllocate({})
@SCJPhase({
   javax.safetycritical.annotate.Phase.STARTUP,
   javax.safetycritical.annotate.Phase.INITIALIZATION,
   javax.safetycritical.annotate.Phase.RUN,
   javax.safetycritical.annotate.Phase.CLEANUP })
public RelativeTime(long millis, int nanos, Chronograph clock)
```

Construct a RelativeTime object representing an interval based on the parameter millis plus the parameter nanos. The construction is subject to millis and nanos parameters normalization. If there is an overflow in the millisecond component when normalizing then an IllegalArgumentException will be thrown.

The clock association is made with the clock parameter.

millis — The desired value for the millisecond component of this. The actual value is the result of parameter normalization.

nanos — The desired value for the nanosecond component of this. The actual value is the result of parameter normalization.

clock — The chronograph providing the association for the newly constructed object. If chronograph is null the association is made with the real-time clock.

Throws IllegalArgumentException if there is an overflow in the millisecond component when normalizing.

```
@SCJAllowed
@SCJMaySelfSuspend(false)
@SCJMayAllocate({})
@SCJPhase({
    javax.safetycritical.annotate.Phase.STARTUP,
    javax.safetycritical.annotate.Phase.INITIALIZATION,
    javax.safetycritical.annotate.Phase.RUN,
    javax.safetycritical.annotate.Phase.CLEANUP })
public RelativeTime( )
```

This constructor behaves the same as calling RelativeTime(0, 0, null).

```
@SCJAllowed
@SCJMaySelfSuspend(false)
@SCJMayAllocate({})
@SCJPhase({
    javax.safetycritical.annotate.Phase.STARTUP,
    javax.safetycritical.annotate.Phase.INITIALIZATION,
    javax.safetycritical.annotate.Phase.RUN,
    javax.safetycritical.annotate.Phase.CLEANUP })
public RelativeTime(long millis, int nanos)
```

This constructor behaves the same as calling RelativeTime(millis, nanos, null).

```
@SCJAllowed
@SCJMaySelfSuspend(false)
@SCJMayAllocate({})
@SCJPhase({
    javax.safetycritical.annotate.Phase.STARTUP,
    javax.safetycritical.annotate.Phase.INITIALIZATION,
    javax.safetycritical.annotate.Phase.RUN,
    javax.safetycritical.annotate.Phase.CLEANUP })
public RelativeTime(Chronograph clock)
```

This constructor behaves the same as calling RelativeTime(0, 0, clock).

```
@SCJAllowed
@SCJMaySelfSuspend(false)
@SCJMayAllocate({javax.safetycritical.annotate.AllocationContext.CURRENT})
@SCJPhase({
    javax.safetycritical.annotate.Phase.STARTUP,
    javax.safetycritical.annotate.Phase.INITIALIZATION,
    javax.safetycritical.annotate.Phase.RUN,
    javax.safetycritical.annotate.Phase.CLEANUP })
public RelativeTime(RelativeTime time)
```

Make a new RelativeTime object from the given RelativeTime object.

The new object will have the same clock association as the time parameter.

time — The RelativeTime object which is the source for the copy.

Throws IllegalArgumentException if the time parameter is null.

Methods

```
@SCJAllowed
@SCJMaySelfSuspend(false)
@SCJMayAllocate({javax.safetycritical.annotate.AllocationContext.CURRENT})
@SCJPhase({
    javax.safetycritical.annotate.Phase.STARTUP,
    javax.safetycritical.annotate.Phase.INITIALIZATION,
    javax.safetycritical.annotate.Phase.RUN,
    javax.safetycritical.annotate.Phase.CLEANUP })
public javax.realtime.RelativeTime add(RelativeTime time)
```

Create a new instance of RelativeTime representing the result of adding time to the value of this and normalizing the result. The clock associated with this and the clock associated with the time parameter are expected to be the same, and such association is used for the result. An IllegalArgumentException is thrown if the clock associated with this and the clock associated with the time parameter are different. An IllegalArgumentException is thrown if the time parameter is null. An ArithmeticException is thrown if the result does not fit in the normalized format.

time — The time to add to this.

returns a new RelativeTime object whose time is the normalization of this plus the parameter time.

Throws IllegalArgumentException if the clock associated with this and the clock associated with the time parameter are different, or when the time parameter is null.

Throws ArithmeticException if the result does not fit in the normalized format.

```
@SCJAllowed
@SCJMaySelfSuspend(false)
@SCJMayAllocate({javax.safetycritical.annotate.AllocationContext.CURRENT})
@SCJPhase({
    javax.safetycritical.annotate.Phase.STARTUP,
    javax.safetycritical.annotate.Phase.INITIALIZATION,
    javax.safetycritical.annotate.Phase.RUN,
    javax.safetycritical.annotate.Phase.CLEANUP })
public javax.realtime.RelativeTime add(RelativeTime time, RelativeTime dest)
```

Return an object containing the value resulting from adding time to the value of this and normalizing the result. If dest is not null, the result is placed there and returned. Otherwise, a new object is allocated for the result. The clock associated with this and the clock associated with the time parameter are expected to be the same, and such association is used for the result. The clock associated with the dest parameter is ignored. An IllegalArgumentException is thrown if the clock associated with this and the clock associated with the time parameter are different. An IllegalArgumentException is thrown if the time parameter is null. An ArithmeticException is thrown if the result does not fit in the normalized format.

time — The time to add to this.

dest — If dest is not null, the result is placed there and returned. Otherwise, a new object is allocated for the result.

returns the value of dest if dest is not null, otherwise the result is returned in a newly allocated object.

Throws IllegalArgumentException if the clock associated with this and the clock associated with the time parameter are different, or when the time parameter is null.

Throws ArithmeticException if the result does not fit in the normalized format.

```
@SCJAllowed
@SCJMaySelfSuspend(false)
@SCJMayAllocate({javax.safetycritical.annotate.AllocationContext.CURRENT})
@SCJPhase({
   javax.safetycritical.annotate.Phase.STARTUP,
   javax.safetycritical.annotate.Phase.INITIALIZATION,
   javax.safetycritical.annotate.Phase.RUN,
   javax.safetycritical.annotate.Phase.CLEANUP })
public javax.realtime.RelativeTime add(long millis, int nanos, RelativeTime dest)
```

Return an object containing the value resulting from adding millis and nanos to the values from this and normalizing the result. If dest is not null, the result is placed there and returned. Otherwise, a new object is allocated for the result. The result will have the same clock association as this, and the clock association with dest is ignored. An ArithmeticException is thrown if the result does not fit in the normalized format.

millis — The number of milliseconds to be added to this.

nanos — The number of nanoseconds to be added to this.

dest — If dest is not null, the result is placed there and returned. Otherwise, a new object is allocated for the result.

returns the value of dest if dest is not null, otherwise a new object representing the result.

Throws ArithmeticException if the result does not fit in the normalized format.

```
@SCJAllowed
@SCJMaySelfSuspend(false)
@SCJMayAllocate({javax.safetycritical.annotate.AllocationContext.CURRENT})
@SCJPhase({
    javax.safetycritical.annotate.Phase.STARTUP,
    javax.safetycritical.annotate.Phase.INITIALIZATION,
    javax.safetycritical.annotate.Phase.RUN,
    javax.safetycritical.annotate.Phase.CLEANUP })
public javax.realtime.RelativeTime add(long millis, int nanos)
```

Create a new object representing the result of adding millis and nanos to the values from this and normalizing the result. The result will have the same clock association as this. An ArithmeticException is thrown if the result does not fit in the normalized format.

millis — The number of milliseconds to be added to this.

nanos — The number of nanoseconds to be added to this.

returns a new RelativeTime object whose time is the normalization of this plus millis and nanos.

Throws ArithmeticException if the result does not fit in the normalized format.

returns A new object containing the result of the addition.

```
@SCJAllowed
@SCJMaySelfSuspend(false)
@SCJMayAllocate({})
@Override
@SCJPhase({
    javax.safetycritical.annotate.Phase.STARTUP,
    javax.safetycritical.annotate.Phase.INITIALIZATION,
    javax.safetycritical.annotate.Phase.RUN,
    javax.safetycritical.annotate.Phase.CLEANUP })
public int compareTo(RelativeTime time)
```

Compares this object with the specified object for order.

returns a negative integer, zero, or a positive integer as this object is less than, equal to, or greater than the specified object.

```
@SCJAllowed
@SCJMaySelfSuspend(false)
@SCJMayAllocate({javax.safetycritical.annotate.AllocationContext.CURRENT})
@SCJPhase({
   javax.safetycritical.annotate.Phase.STARTUP,
   javax.safetycritical.annotate.Phase.INITIALIZATION,
   javax.safetycritical.annotate.Phase.RUN,
   javax.safetycritical.annotate.Phase.CLEANUP })
public javax.realtime.RelativeTime subtract(RelativeTime time, RelativeTime dest)
```

Return an object containing the value resulting from subtracting the value of time from the value of this and normalizing the result. If dest is not null, the result is placed there and returned. Otherwise, a new object is allocated for the result. The clock associated with this and the clock associated with the time parameter are expected to be the same, and such association is used for the result. The clock associated with the dest parameter is ignored. An IllegalArgumentException is thrown if the clock associated with this and the clock associated with the time parameter are different. An IllegalArgumentException is thrown if the time parameter is null. An ArithmeticException is thrown if the result does not fit in the normalized format.

time — The time to subtract from this.

dest — If dest is not null, the result is placed there and returned. Otherwise, a new object is allocated for the result.

returns the value of dest if dest is not null, otherwise a new object representing the result.

Throws IllegalArgumentException if the if the clock associated with this and the clock associated with the time parameter are different, or when the time parameter is null.

Throws ArithmeticException if the result does not fit in the normalized format.

```
@SCJAllowed
@SCJMaySelfSuspend(false)
@SCJMayAllocate({javax.safetycritical.annotate.AllocationContext.CURRENT})
@SCJPhase({
   javax.safetycritical.annotate.Phase.STARTUP,
   javax.safetycritical.annotate.Phase.INITIALIZATION,
   javax.safetycritical.annotate.Phase.RUN,
   javax.safetycritical.annotate.Phase.CLEANUP })
public javax.realtime.RelativeTime subtract(RelativeTime time)
```

Create a new instance of RelativeTime representing the result of subtracting time from the value of this and normalizing the result. The clock associated

with this and the clock associated with the time parameter are expected to be the same, and such association is used for the result. An IllegalArgumentException is thrown if the clock associated with this and the clock associated with the time parameter are different. An IllegalArgumentException is thrown if the time parameter is null. An ArithmeticException is thrown if the result does not fit in the normalized format.

time — The time to subtract from this.

returns a new RelativeTime object whose time is the normalization of this minus the parameter time parameter time.

Throws IllegalArgumentException if the clock associated with this and the clock associated with the time parameter are different, or when the time parameter is null.

Throws ArithmeticException if the result does not fit in the normalized format.

8.5 Rationale

Many SCJ systems do not have access to a time synchronization service or the current date. Therefore, SCJ does not require any particular *Epoch*. On a system without the notion of calendar time AbsoluteTime(0,0) may represent the system start up time.

As time values from different chronographs are not comparable, comparison of time values from different clocks is not supported by SCJ.

The concept of requiring times (e.g., HighResolutionTime) to be immutable was considered, but further consideration showed that its implementation would be difficult without generating excessive garbage. As a result, it was decided that times used in SCJ applications would continue to be mutable as they are in the RTSJ.

8.6 Compatibility

The RTSJ defines the Epoch to be the 1st day of January 1970, but this Epoch is not required in SCJ.

Chapter 9

Java Metadata Annotations

9.1 Overview

This chapter describes Java Metadata annotations used by the SCJ. Java Metadata annotations enable developers to add additional typing information to a Java program, thereby enabling more detailed functional and non functional analyses, both for ensuring program consistency and for aiding the runtime system to produce more efficient code. They are retained in the compiled bytecode intermediate format and are thus available for performing validation at class load-time. In SCJ, an important use of metadata annotations is to enforce compliance levels and to restrict the behavior of certain methods.

This specification distinguishes between *application code* and *infrastructure code*. It is strongly recommended that implementations should provide a *Checker* tool to ensure that the application code abides by the restrictions defined by its annotations. Infrastructure code shall be verified by the vendor. Infrastructure code includes the java and javax packages as well as vendor specific libraries.

The Overview, as well as the Rationale and Examples sections, are not normative but are provided to improve understanding of the normative sections. All of the other sections of this chapter are normative.

9.2 Semantics and Requirements

The SCJ annotations described in this chapter address the following two groups of properties:

> *Compliance Levels* — The SCJ specification defines three levels of compliance. Both application and infrastructure code must adhere to one of these

compliance levels. Consequently, code belonging to a certain level may access only code that is at the same or lower level. This ensures that an SCJ application is compatible with the SCJ infrastructure and other application code with respect to the specified SCJ level.

Behavioral Restrictions — Because each mission consists of a sequence of specific phases (i.e., start up, initialization, execution, clean up), the application must clearly distinguish between these phases. Furthermore, it is illegal for the application to access SCJ functionality that is not provided for its current execution phase within a mission.

9.3 Annotations for Enforcing Compliance Levels

API visibility annotations are used to prevent application programmers from accessing SCJ API methods that are intended to be internal.

The SCJ specification specifies three compliance levels to which applications and implementations shall conform. Each level specifies restrictions on what APIs are permitted for use by an application, with lower levels strictly more restrictive than higher levels. The @SCJAllowed() metadata annotation is introduced to indicate the compliance level of classes and members. The @SCJAllowed() annotation is summarized in Table 9.1 and takes two arguments.

Annotation	Argument	Values	Description
@SCJAllowed	value	LEVEL_0 LEVEL_1 LEVEL_2	Application-level.
		SUPPORT	Application-level, accessed by library.
	members	TRUE FALSE	Inherit value by sub-elements.

Table 9.1: Compliance LEVEL annotation.

1. The default argument of type Level specifies the level of the annotation target. The options are LEVEL_0, LEVEL_1, LEVEL_2, and SUPPORT.

   - Level 0, Level 1, and Level 2 specify that an element may only be visible by those elements that are at the specified level or higher. Therefore, a method that is @SCJAllowed(LEVEL_2) may invoke a method that is @SCJAllowed(LEVEL_1), but not vice versa. In addition, a method annotated with a certain level may not have a higher level than a method that it overrides.

- SUPPORT specifies an application-level method that can be invoked only by the infrastructure code; the SUPPORT annotation cannot be used to specify a level of a class. A SUPPORT method cannot be invoked by other SUPPORT methods. A SUPPORT method can invoke other application-level methods up to the level specified by its enclosing class. This implies that a SUPPORT method acts as if its level were set to the level of its enclosing class.

The default value when no value is specified is LEVEL_0. When no annotation applies to a class or member it is not visible. The ordering on annotations is LEVEL_0 $<$ LEVEL_1 $<$ LEVEL_2 $<$ SUPPORT.

2. The second argument, members, determines whether or not the specified compliance level recurses to nested members and classes. The default value is false.

9.3.1 Compliance Level Reasoning

The compliance level of a class or member shall be the first of the following:

1. The level specified on its own @SCJAllowed() annotation, if it exists,
2. The level of the closest outer element with an @SCJAllowed() annotation, if members = true,

If a class, interface, or member has compliance level C, it shall be used in code that also has compliance level C or higher. It is legal for an implementation to not emit code for methods and classes that may not be used at the chosen level of an SCJ application, though it may be necessary to provide stubs in certain cases.

It is illegal for an overriding method to change the compliance level of the overridden method. It is also illegal for a subclass to have a lower compliance level than its superclass. Each element shall either correctly override the @SCJAllowed annotation of the parent or restate the parent's annotation. All enclosed elements of a class or member shall have a compliance level greater than or equal to the enclosing element.

Methods annotated SUPPORT may be overridden by the application and if so, the SUPPORT annotation must be restated.

Static initializers have the same compliance level as their defining class, regardless of the members argument.

9.3.2 Class Constructor Rules

For a class that is annotated @SCJAllowed with members = true, all constructors shall default to @SCJAllowed at the same compliance level.

If a class has a default constructor, the constructor's compliance level shall be that of the class if the annotation has members = true.

9.3.3 Other Rules

Exceptions thrown by a method must be visible at the compliance level of that method.

9.4 Annotations for Restricting Behavior

The following set of annotations is provided to express some dynamic characteristics of methods. For example, some methods may be restricted from memory allocation, some methods may only be called in a certain mission phase, or may be restricted from using blocking calls. For these situations, the restricted behavior annotations @SCJMayAllocate, @SCJMaySelfSuspend, and @SCJPhase are used.

The restricted behavior annotations may be applied to a class, interface, or enumeration to change the default values for the methods on that class, interface, or enumeration.

9.4.1 @SCJMayAllocate

The @SCJMayAllocate annotation takes an array of the Allocation enumeration. When a method is annotated with a @SCJMayAllocate array that is not empty, the annotated method is allowed to perform allocation or call methods with a consistent annotation. If a method is annotated with an empty @SCJMayAllocate array, then all methods that override it must also be annotated with an empty SCJMayAllocate array. Methods that are annotated @SCJMayAllocate(CurrentContext) may contain expressions that result in allocation in the current scope (e.g. at the source level new expressions, string concatenation, and autoboxing). Methods that are annotated @SCJMayAllocate(OuterContext) may contain expressions that result in allocation in an outer scope within which the current scope is nested. Methods that are annotated @SCJMayAllocate (InnerContext) may contain expressions that result in allocation in a scope nested within the current scope. Methods that are annotated @SCJMayAllocate(ThisContext) may contain expressions that result in allocation in the scope containing this.

If no @SCJMayAllocate annotation is present, the default value for @SCJMayAllocate is CurrentContext, OuterContext, Inner-Context.

9.4.2 @SCJMaySelfSuspend

The @SCJMaySelfSuspend annotation is boolean and takes values of true and false. When @SCJMaySelfSuspend is true, the annotated method may take an action that causes it to block. If a method is marked @SCJMaySelfSuspend(false), then neither it nor any method it calls may take an action causing it to block.

The default value is true.

9.4.3 @SCJPhase

The @SCJPhase annotation takes an array of the Phase enumeration. The SCJ application phases are: STARTUP, INITIALIZATION, RUN, and CLEANUP.

The default phases for all methods without a phase annotation is that the method can be executed in any of the four phases.

9.4.4 Inheritance Considerations

When a method that is coded with any of these restricted behavior annotations is overridden, the overriding method shall be annotated with the parent's restricted behavior annotations or it shall have appropriate restricted behavior annotations that are consistent with its parent method with respect to the parent's @SCJMayAllocate, @SCJMaySelfSuspend, and @SCJPhase settings:

- For the @SCJMayAllocate annotation, the consistency requirement for overriding child methods means that the child's annotations may be more restricted than the parent's annotations, but shall not be less restricted than the parent's annotations. For example, if the parent uses the annotation @SCJMayAllocate({CurrentContext, InnerContext}), the child shall not incorporate the annotation @SCJMayAllocate({OuterContext}) because such a child annotation would violate the annotation of the parent. However, the child could use the annotation @SCJMayAllocate(CurrentContext) because it is more restrictive than the parent. With respect to @SCJMayAllocate, the following is a list of the annotations in order of decreasing restriction: No allocation (i.e., {}), CurrentContext, ThisContext, InnerContext, and OuterContext.
- If a method with a @SCJMaySelfSuspend annotation is overridden, the overriding method shall inherit the same setting for @SCJMaySelfSuspend as its

parent. For the @SCJMaySelfSuspend annotation, the consistency requirement for overriding child methods means that the child's annotations may be more restricted than the parent's annotations, but shall not be less restricted than the parent's annotations. For example, if the parent incorporates the annotation @SCJMaySelfSuspend(false), the child shall not incorporate the annotation @SCJMaySelfSuspend(true) because it is less restrictive to allow self-suspension, but the converse is a valid annotation.

- For the @SCJPhase annotation, the consistency requirement for overriding child methods means that the child's annotations must be exactly the same as the parent's annotations.

9.5 Level Considerations

These annotations apply to all levels.

9.6 API

9.6.1 Class javax.safetycritical.annotate.SCJPhase

Declaration

```
@SCJAllowed
@Retention(java.lang.annotation.RetentionPolicy.CLASS)
@Target({
   java.lang.annotation.ElementType.TYPE, java.lang.annotation.ElementType.METHOD,
   java.lang.annotation.ElementType.CONSTRUCTOR})
public @interface SCJPhase
```

Description

This annotation distinguishes methods that may be called only from code running in a certain mission phase (e.g. Initialization or Clean Up).

Attributes

```
@SCJAllowed
public javax.safetycritical.annotate.Phase [] value () default {
javax.safetycritical.annotate.Phase.STARTUP,
javax.safetycritical.annotate.Phase.INITIALIZATION,
javax.safetycritical.annotate.Phase.RUN,
javax.safetycritical.annotate.Phase.CLEANUP };
```

The phase of the mission in which a method may run.

9.6.2 Class javax.safetycritical.annotate.SCJMayAllocate

Declaration

```
@SCJAllowed
@Retention(java.lang.annotation.RetentionPolicy.CLASS)
@Target({
   java.lang.annotation.ElementType.TYPE, java.lang.annotation.ElementType.METHOD,
   java.lang.annotation.ElementType.CONSTRUCTOR})
public @interface SCJMayAllocate
```

Description

This annotation distinguishes methods that may be restricted from allocating memory in certain memory areas.

Attributes

```
@SCJAllowed
public javax.safetycritical.annotate.AllocationContext [] value () default {
javax.safetycritical.annotate.AllocationContext.CURRENT,
javax.safetycritical.annotate.AllocationContext.OUTER,
javax.safetycritical.annotate.AllocationContext.INNER };
```

9.6.3 Class javax.safetycritical.annotate.SCJMaySelfSuspend

Declaration

```
@SCJAllowed
@Retention(java.lang.annotation.RetentionPolicy.CLASS)
@Target({
   java.lang.annotation.ElementType.TYPE, java.lang.annotation.ElementType.METHOD,
   java.lang.annotation.ElementType.CONSTRUCTOR})
public @interface SCJMaySelfSuspend
```

Description

This annotation distinguishes methods that may be restricted from blocking during execution.

Attributes

```
@SCJAllowed
public boolean value () default false;
```

9.6.4 Class javax.safetycritical.annotate.SCJAllowed

Declaration

```
@SCJAllowed
@Retention(java.lang.annotation.RetentionPolicy.CLASS)
@Target({
   java.lang.annotation.ElementType.TYPE, java.lang.annotation.ElementType.FIELD,
   java.lang.annotation.ElementType.METHOD,
   java.lang.annotation.ElementType.CONSTRUCTOR})
public @interface SCJAllowed
```

Description

This annotation distinguishes methods, classes, and fields that may be accessed from within safety-critical Java programs. In some implementations of the safety-critical Java specification, elements which are not declared with the @SCJAllowed annotation (and are therefore not allowed in safety-critical application software) are present within the declared class hierarchy. These are necessary for full compatibility with standard edition Java, the Real-Time Specification for Java, and/or for use by the implementation of infrastructure software.

The value field equals LEVEL_0 for elements that may be used within safety-critical Java applications targeting Level 0, Level 1, or Level 2.

The value field equals LEVEL_1 for elements that may be used within safety-critical Java applications targeting Level 1 or Level 2.

The value field equals LEVEL_2 for elements that may be used within safety-critical Java applications targeting Level 2.

Absence of this annotation on a given Class, Field, Method, or Constructor declaration indicates that the corresponding element may not be accessed from within a compliant safety-critical Java application.

Attributes

```
@SCJAllowed
public boolean members () default false;
```

```
@SCJAllowed
public javax.safetycritical.annotate.Level value () default
javax.safetycritical.annotate.Level.LEVEL_0;
```

9.6.5 Class javax.safetycritical.annotate.Level

Declaration

```
@SCJAllowed
public enum Level
```

Fields

@SCJAllowed
public static final javax.safetycritical.annotate.Level LEVEL_0

@SCJAllowed
public static final javax.safetycritical.annotate.Level LEVEL_1

@SCJAllowed
public static final javax.safetycritical.annotate.Level LEVEL_2

@SCJAllowed
public static final javax.safetycritical.annotate.Level SUPPORT

9.6.6 Class javax.safetycritical.annotate.Phase

Declaration

@SCJAllowed
public enum Phase

Fields

@SCJAllowed
public static final javax.safetycritical.annotate.Phase CLEANUP

@SCJAllowed
public static final javax.safetycritical.annotate.Phase INITIALIZATION

@SCJAllowed
public static final javax.safetycritical.annotate.Phase RUN

@SCJAllowed
public static final javax.safetycritical.annotate.Phase STARTUP

9.6.7 Class javax.safetycritical.annotate.AllocationContext

Declaration

@SCJAllowed
public enum AllocationContext

Fields

@SCJAllowed
public static final javax.safetycritical.annotate.AllocationContext CURRENT

Allocation is allowed in the current memory area.

@SCJAllowed
public static final javax.safetycritical.annotate.AllocationContext INNER

Allocation is allowed in any inner (more deeply nested) memory area.

@SCJAllowed
public static final javax.safetycritical.annotate.AllocationContext OUTER

Allocation is allowed in any outer (less deeply nested) memory area.

@SCJAllowed
public static final javax.safetycritical.annotate.AllocationContext **THIS**

Allocation is allowed in the memory area where the current object (this) was allocated.

9.7 Rationale and Examples

It is expected that the metadata annotations will be checked by analysis tools as well as at load time (or link time if class loading is integrated with the linking). Virtual machines that use an ahead-of-time compilation model are expected to perform the checks when the executable image of the program is assembled.

9.7.1 Compliance Level Annotation Example

The following example illustrates an application of the compliance level annotation. The example shows both application and infrastructure fragments of source code, demonstrating the application of the compliance level annotations.

```
@SCJAllowed(LEVEL_0, members=true)
class MyMission extends CyclicExecutive {

    WordHandler peh;

    @SCJAllowed(SUPPORT)
```

```
    public void initialize() {
        peh = new MyHandler(...); // ERROR – because MyHandler is Level 1
    }
}

@SCJAllowed(LEVEL_1)
class MyHandler extends PeriodicEventHandler {

    @SCJAllowed(SUPPORT)
    public void handleAsyncEvent() {...}
}

@SCJAllowed(LEVEL_0)
public abstract class PeriodicEventHandler extends ManagedEventHandler {

  @SCJAllowed(LEVEL_0)
  public PeriodicEventHandler(..) {...}

  @SCJAllowed(LEVEL_0) // ERROR – because getActualStartTime is Level 1
  public HighResolutionTime getActualStartTime() {...}
}
```

It is evident that all the elements of the example are declared to reside at a specific compliance level. At the application domain, class MyMission is declared to be at Level 0. Every Level 0 mission is composed of one or more periodic handlers; in this case, we define the MyHandler class. The handler is, however, declared to be at Level 1, which is an error.

Looking at the SCJ infrastructure code, the PeriodicEventHandler class implements the Schedulable interface, both of which are defined as Level 0 compliant. However, PeriodicEventHandler is defined to override getActualStartTime(), originally allowed only at Level 1. This results in an illegal attempt to increase method visibility.

9.7.2 Memory Safety Annotations

In an early draft of this SCJ specification, the JSR-302 Expert Group considered an expanded set of annotations designed to ensure the safety of memory references in SCJ applications. However, the SCJ Expert Group later determined that those proposed memory safety annotations were not ready for standardization, so they were removed. The Expert Group expects that a later extension to this specification will consider expanding these annotations to address memory safety.

Chapter 10

Java Native Interface

10.1 Overview

SCJ provides more restrictions than the RTSJ to simplify the run-time infrastructure and assist with safety-critical analysis for certification. This chapter defines these additional SCJ restrictions.

The Overview, Rationale, and Example sections are not normative but are provided to improve understanding of the normative sections. All of the other sections of this chapter are normative. It is important that SCJ implementations shall follow the common JNI rules found (for Java 8.0) at **http://docs.oracle.com/javase/8/docs/technotes/guides/jni/index.html**, while obeying the restrictions in the normative sections of this chapter.

10.2 Semantics and Requirements

If the underlying run-time infrastructure supports native code execution, then all Java native interface (JNI) supported functions described in this chapter shall be implemented; otherwise, JNI is not available to the application.

10.3 Level Considerations

Due to SCJ limitations concerning reflection and object allocation the JNI support is restricted to a basic functionality that can be used equally for native methods on Level 0, Level 1, and Level 2.

10.4 API

This section follows the structure of Chapter 4 in the common guide to JNI. For each of the subsections, there is a list of functions that shall be supported, and when relevant, a list of functions that need not be supported, because they are in conflict with SCJ restrictions.

The effect of an application referencing the unsupported JNI functions defined by standard Java is implementation defined. The possible effects include, but are not limited to: compile time error, a null pointer, or a function aborting execution. The recommended implementation is to not provide definitions for any unsupported functions.

Developers of SCJ applications should note the Exception capabilities specified in Chapter 11 that make it possible to throw pre-allocated exceptions without causing potential memory leaks.

10.4.1 Version Information

The function to retrieve the JNI version shall be supported : GetVersion()

10.4.2 Class Operations

Two class inspection functions shall be supported: GetSuperclass() and IsAssignableFrom()

The following functions are NOT required to be supported, because they require reflection and/or dynamic class loading to operate: DefineClass() and FindClass().

10.4.3 Exceptions

The functions Throw(), ExceptionOccurred(), ExceptionClear(), FatalError() and ExceptionCheck() shall be supported.

The following functions are NOT required to be supported, because they require object allocation or IO to operate: ThrowNew() and ExceptionDescribe().

10.4.4 Global and Local References

The following functions shall be supported because their implementation is believed to present no certifiability problems, and provide SCJ applications with a high degree

of compatibility with existing JNI code: EnsureLocalCapacity(), PushLocalFrame(), PopLocalFrame(), and NewLocalRef().

The following function are NOT required to be supported, because their semantics conflict with scoped memory, and require features (like weak references) not found in an SCJ implementation: NewGlobalRef(), DeleteGlobalRef() and DeleteLocalRef().

10.4.5 Weak Global References

Functions for weak global references are NOT required to be supported, because their semantics conflict with scoped memory and require features not found in an SCJ implementation: NewWeakGlobalRef() and DeleteWeakGlobalRef().

10.4.6 Object Operations

The following functions provide basic operations on objects and require no reflection and no object allocation.They shall be supported: GetObjectClass(), GetObjectRefType(), IsInstanceOf(), IsSameObject().

The following functions are NOT required to be supported because they require reflection: AllocObject(), NewObject(), NewObjectA() and NewObjectV().

10.4.7 Accessing Fields of Objects

These functions are NOT required to be supported because they require reflection: GetFieldID(), Get<type>Field() and Set<type>Field().

10.4.8 Calling Instance Methods

These functions are NOT required to be supported because they require reflection: GetMethodID(), Call<type>Method(), Call<type>MethodA(), Call<type>MethodV(), as well as these functions: CallNonvirtual<type>Method(), CallNonvirtual<type>MethodA() and CallNonvirtual<type>MethodV().

10.4.9 Accessing Static Fields

The functions are NOT required to be supported because they require reflection: GetStaticFieldID(), GetStatic<type>Field() and SetStatic<type>Field().

10.4.10 Calling Static Methods

These functions are NOT required to be supported because they require reflection: GetStaticMethodID(), CallStatic<type>Method(), CallStatic<type>MethodA() and CallStatic<type>MethodV().

10.4.11 String Operations

The following functions shall be supported because they are basic operations on strings and require no allocation: GetStringLength(), GetStringUTFLength(), GetStringRegion() and GetStringUTFRegion().

The following functions are NOT required to be supported because they may require allocation: NewString(), GetStringChars(), ReleaseStringChars(), NewStringUTF(), GetStringUTFChars(), as well as ReleaseStringUTFChars(), GetStringCritical() and ReleaseStringCritical().

10.4.12 Array Operations

The following basic functions require no allocation and shall be supported: GetArrayLength(),as well as GetObjectArrayElement() and SetObjectArrayElement().

The following functions are NOT required to be supported because they require allocation:
NewObjectArray(), New<PrimitiveType>Array(),
Get<PrimitiveType>ArrayElements(), Release<PrimitiveType>ArrayElements(),
Get<PrimitiveType>ArrayRegion(), Set<PrimitiveType>ArrayRegion(),
GetPrimitiveArrayCritical() and ReleasePrimitiveArrayCritical().

10.4.13 Registering Native Methods

The RegisterNatives() function shall be required to be supported, although it shall not be called after return from Safelet.initializeApplication. It is needed to disambiguate between the two possible naming conventions for JNI functions in systems where the Java implementation does not control linking.

The UnregisterNatives() function is NOT required to be supported because it is only useful for systems with dynamic loading.

10.4.14 Monitor Operations

The MonitorEnter() and MonitorExit() functions are NOT required to be supported because they map to synchronized() which is restricted in SCJ.

10.4.15 NIO Support

In SCJ support is NOT required for native IO functions: NewDirectByteBuffer(), GetDirectBufferAddress(), GetDirectBufferCapacity(),

10.4.16 Reflection Support

In SCJ support is NOT required for reflection functions: FromReflectedMethod(), FromReflectedField(), ToReflectedMethod() and ToReflectedField().

10.4.17 Java VM Interface

In SCJ support is NOT required for VM Interface functions.

10.5 Annotations

There is no SCJ support for verification of the annotations of native methods. On the other hand, it is important to provide this information to any available tools for validating SCJ programs for correctness and safety. To ensure that application programmers consider their implementation carefully, there are no default annotations for native methods concerning allocation and blocking. Therefore it is always required to annotate native methods with the @SCJMayAllocate and@SCJMaySelfSuspend annotations.

If the native method does not call back to Java methods, its @SCJMayAllocate attribute shall contain an empty array or {CurrentContext}, and it shall also be annotated with an appropriate setting for @SCJMaySelfSuspend. If the annotation @SCJMayAllocate({CurrentContext} is specified, it indicates that the native method allocates native memory dynamically. SCJ compliant implementations of native methods shall not allocate objects in SCJ memory.

If the native method calls back to one or more Java methods, the annotation of the native method should be consistent with any Java methods that may be called.

Note that any JNI code that may be called within a synchronized method shall be annotated as @SCJMaySelfSuspend(false) to indicate that it will never self-suspend.

10.6 Rationale

Due to the complexity of static analysis of code that contains reflection, the SCJ restricts all use of reflection and object allocation at all levels. As such, many of the functions that would normally be available in JNI are not supported. In addition, no functions that require allocation will be required for SCJ conformance.

Call-back functions from C to create, attach or unload the JVM are not required because the corresponding operations are not supported.

It is difficult for a Java standard such as SCJ to prescribe how native code must be supported. Instead SCJ recommends not including any definitions of unsupported functions to enable early detection of references.

10.7 Example

```
@SCJAllowed
@SCJMayAllocate({})
@SCJMaySelfSuspend(false)
static native int getProcessorId(String theProcessorInformationString);
```

The native method is called with a previously allocated string as parameter. Besides the integer return value, in this example, the parameter of type string can be used to return information to the Java context. Because it is marked @SCJMayAllocate({}), the implementation of getProcessorId must not allocate memory dynamically. Because the desired information might be obtained by a call to the operation system, maySelfSuspend=false is used.

Header files of the native implementation can be generated by javah as usual.

10.8 Compatibility

10.8.1 RTSJ Compatibility Issues

The restrictions in Level 0 are upwardly compatible with a conformant RTSJ solution in that, applications that will run under this restricted environment will also work correctly under a less restricted environment such as CLDC or JSE.

This will not affect standard RTSJ applications, unless they are using JNI functions that are not supported.

For consistency with standard RTSJ applications, if an SCJ implementation supports allocation of Java objects from native code, such allocations should allocate objects using the current allocation context at the point of the call.

10.8.2 General Java Compatibility Issues

Existing JNI code may need to be modified for use in an SCJ application due to the reduced set of JNI functions that are supported for SCJ. In particular, to modify fields of an object, the field will need to be passed as an argument to the underlying JNI function because there is no way to access a field directly.

Chapter 11

Exceptions

11.1 Overview

Exceptions are used in Java as well as other languages to separate functional logic from error handling. Safety-critical applications in languages such as Ada and C++, however, usually avoid their use. One reason for avoiding their use is that exception propagation may introduce execution paths for which execution time is difficult to analyze.

In Java, it is impossible to avoid exception handlers altogether due to the existence of checked exceptions which can be thrown by many standard methods. Compiler analysis such as dataflow analysis can help ensure that certain exceptions will never be thrown by a given method invocation, but in general it is not possible to eliminate the execution of all throw statements and catch clauses.

This chapter describes how exceptions can be thrown and caught within SCJ programs while avoiding memory leaks or out-of-memory exceptions. It is expected that observing these principles will permit safe exception handling within infrastructure classes as well as application classes.

The Overview and Rationale sections are not normative but are provided to improve understanding of the normative sections. All of the other sections of this chapter are normative.

11.2 Semantics and Requirements

Except for how information associated with a Throwable is stored and managed, the semantics of the subclasses of Error, Exception, and RuntimeException are the same as for all other Java throwables.

Like the RTSJ, SCJ also supports statically allocated exceptions for those exceptions that implement the RTSJ interface StaticThrowable.There is at most one instance of each of these exceptions and errors, managed by the runtime. The message and stack information they would normally carry is held in a thread-local data structure. This means this information is only valid within the context of the thread that threw the StaticThrowable, and there only until a new StaticThrowable is thrown. For these static exception, an allocation should not be used. Instead an exception is thrown by calling the static getMethod associated with the StaticThrowable.

The thread-local storage used by StaticThrowables is controlled by the ConfigurationParameters associated with the active ManagedSchedulable when the exception is thrown. See Section 4.4.7.

It is implementation-defined how a particular implementation of SCJ captures and represents thread backtraces for thrown exceptions.

Exception object allocation through the keyword new uses the current allocation context. Throw statements and catch clauses work the same in SCJ as in RTSJ. There are no special requirements on checked or unchecked exceptions.

An attempt to propagate an exception out of its scope (i.e. out of the ScopedMemory in which it is allocated) is called a *boundary error*. The exception which causes a boundary error is called the *original exception*. A boundary error stops the propagation of the original exception and throws a ThrowBoundaryError exception in its place (as in RTSJ). SCJ defines its own ThrowBoundaryError class in javax.safetycritical which extends the corresponding ThrowBoundaryError class of the RTSJ.

**

11.2.1 SCJ-Specific Functionality

A ThrowBoundaryError exception shall contain information about the original exception. This information can be extracted from the most recent boundary error in the current schedulable object using the methods in javax.safetycritical.ThrowBoundaryError.

When SCJ replaces a thrown exception with a ThrowBoundaryError exception, it preserves a reference to the class of the originally thrown exception within the thread-local ThrowBoundaryError object. Whether stack backtrace information is copied at this same time is implementation-defined.

The method getPropagatedMessage returns the message associated with the original exception. The message shall begin with the fully-qualified name of the exception class. The message is truncated by discarding the highest indices if it exceeds the maximum allowed length for this Schedulable object. The method getPropagatedStackTraceDepth returns the number of valid elements in the StackTraceElement ar-

ray returned by getPropagatedStackTrace(). The method getPropagatedStackTrace returns the stack trace copied from the original exception. The stack trace is truncated by discarding the oldest stack trace elements if it exceeds the maximum allowed length for this schedulable object.

11.3 Level Considerations

The support for exceptions is the same for all compliance levels. A method annotated with a particular compliance level shall neither declare nor throw exceptions which have a higher compliance level.

11.4 API

The classes Error and Exception in java.lang provide the same constructors and methods in SCJ as in standard Java. The class Throwable in java.lang provides the same constructors in SCJ as in standard Java; the available methods are restricted in SCJ as described below. The RTSJ exception are used as defined by the RTSJ

Unless indicated otherwise, the classes defined in this section are thread safe

11.4.1 Class java.lang.Throwable

Declaration

```
@SCJAllowed
 public class Throwable implements java.io.Serializable extends java.lang.Object
```

Description

The Throwable class is the superclass of all errors and exceptions in the Java language. Only objects that are instances of this class (or one of its subclasses) are thrown by the Java Virtual Machine or can be thrown by the Java throw statement. Similarly, only this class or one of its subclasses can be the argument type in a catch clause. For the purposes of compile-time checking of exceptions, Throwable and any subclass of Throwable that is not also a subclass of either RuntimeException or Error are regarded as checked exceptions.

Constructors

```
@SCJAllowed
@SCJMaySelfSuspend(false)
@SCJMayAllocate({javax.safetycritical.annotate.AllocationContext.CURRENT})
```

```
@SCJPhase({
    javax.safetycritical.annotate.Phase.STARTUP,
    javax.safetycritical.annotate.Phase.INITIALIZATION,
    javax.safetycritical.annotate.Phase.RUN,
    javax.safetycritical.annotate.Phase.CLEANUP })
public Throwable(String msg, Throwable cause)
```

Constructs a Throwable object with an optional detail message and an optional cause. If cause is null, Services.captureStackBacktrace(this) is called to save the backtrace associated with the current thread. If cause is not null, Services.captureStackBacktrace(this) is not called to avoid overwriting the backtrace associated with the cause.

msg — the detail message for this Throwable object.

cause — the cause of this exception.

```
@SCJAllowed
@SCJMaySelfSuspend(false)
@SCJMayAllocate({
    javax.safetycritical.annotate.AllocationContext.CURRENT,
    javax.safetycritical.annotate.AllocationContext.INNER,
    javax.safetycritical.annotate.AllocationContext.OUTER})
@SCJPhase({
    javax.safetycritical.annotate.Phase.STARTUP,
    javax.safetycritical.annotate.Phase.INITIALIZATION,
    javax.safetycritical.annotate.Phase.RUN,
    javax.safetycritical.annotate.Phase.CLEANUP })
public Throwable( )
```

This constructor behaves the same as calling Throwable(String, Throwable) with the arguments (null, null).

```
@SCJAllowed
@SCJMaySelfSuspend(false)
@SCJMayAllocate({javax.safetycritical.annotate.AllocationContext.CURRENT})
@SCJPhase({
    javax.safetycritical.annotate.Phase.STARTUP,
    javax.safetycritical.annotate.Phase.INITIALIZATION,
    javax.safetycritical.annotate.Phase.RUN,
    javax.safetycritical.annotate.Phase.CLEANUP })
public Throwable(Throwable cause)
```

This constructor behaves the same as calling Throwable(String, Throwable) with the arguments (null, cause).

```
@SCJAllowed
@SCJMaySelfSuspend(false)
@SCJMayAllocate({javax.safetycritical.annotate.AllocationContext.CURRENT})
@SCJPhase({
   javax.safetycritical.annotate.Phase.STARTUP,
   javax.safetycritical.annotate.Phase.INITIALIZATION,
   javax.safetycritical.annotate.Phase.RUN,
   javax.safetycritical.annotate.Phase.CLEANUP })
public Throwable(String msg)
```

This constructor behaves the same as calling Throwable(String, Throwable) with the arguments (msg, null).

Methods

```
@SCJAllowed
@SCJMaySelfSuspend(false)
@SCJMayAllocate({})
@SCJPhase({
   javax.safetycritical.annotate.Phase.STARTUP,
   javax.safetycritical.annotate.Phase.INITIALIZATION,
   javax.safetycritical.annotate.Phase.RUN,
   javax.safetycritical.annotate.Phase.CLEANUP })
public java.lang.Throwable getCause( )
```

returns a reference to the cause that was supplied as an argument to the constructor, or null if no cause was specified at construction time. Performs no memory allocation.

```
@SCJAllowed
@SCJMaySelfSuspend(false)
@SCJMayAllocate({})
@SCJPhase({
   javax.safetycritical.annotate.Phase.STARTUP,
   javax.safetycritical.annotate.Phase.INITIALIZATION,
   javax.safetycritical.annotate.Phase.RUN,
   javax.safetycritical.annotate.Phase.CLEANUP })
public java.lang.String getMessage( )
```

returns a reference to the message that was supplied as an argument to the constructor, or null if no message was specified at construction time. Performs no memory allocation.

```
@SCJAllowed
@SCJMaySelfSuspend(false)
@SCJMayAllocate({javax.safetycritical.annotate.AllocationContext.CURRENT})
@SCJPhase({
   javax.safetycritical.annotate.Phase.STARTUP,
```

```
    javax.safetycritical.annotate.Phase.INITIALIZATION,
    javax.safetycritical.annotate.Phase.RUN,
    javax.safetycritical.annotate.Phase.CLEANUP })
public java.lang.StackTraceElement[] getStackTrace( )
```

Allocates a StackTraceElement array, StackTraceElement objects, and all internal structure, including String objects referenced from each StackTraceElement to represent the stack backtrace information available for the exception that was most recently associated with this Throwable object.

Each Schedulable maintains a single buffer to contain the stack backtrace information associated with the most recent invocation of System.captureStackBacktrace. The size of this buffer is specified by providing a SchedulableSizingParameters object as an argument to construction of the Schedulable. Most commonly, Services.captureStackBacktrace is invoked from within the constructor of java.lang.Throwable . getStackTrace returns the contents of this single backtrace buffer information.

If Services.captureStackBacktrace has been invoked within this thread more recently than the construction of this Throwable, then the stack trace information returned from this method may not represent the stack backtrace for this particular Throwable.

```
@SCJMayAllocate({javax.safetycritical.annotate.AllocationContext.CURRENT})
@SCJAllowed
@SCJMaySelfSuspend(false)
@SCJPhase({
    javax.safetycritical.annotate.Phase.STARTUP,
    javax.safetycritical.annotate.Phase.INITIALIZATION,
    javax.safetycritical.annotate.Phase.RUN,
    javax.safetycritical.annotate.Phase.CLEANUP })
public void printStackTrace(PrintStream stream)
```

Print the stack trace of this Throwable to the given stream.

The printed stack trace contains the result of toString() as the first line followed by one line for each stack trace element that contains the name of the method or constructor, optionally followed by the source file name and source file line number when available.

stream — the stream to print to.

```
@SCJMayAllocate({javax.safetycritical.annotate.AllocationContext.CURRENT})
@SCJAllowed
@SCJMaySelfSuspend(false)
@SCJPhase({
```

```
    javax.safetycritical.annotate.Phase.STARTUP,
    javax.safetycritical.annotate.Phase.INITIALIZATION,
    javax.safetycritical.annotate.Phase.RUN,
    javax.safetycritical.annotate.Phase.CLEANUP })
public void printStackTrace( )
```

11.4.2 Class javax.realtime.StaticThrowable

Declaration

```
@SCJAllowed
 public interface StaticThrowable
```

See Also: javax.realtime.ConfigurationParameters

Description

A marker interface to indicate that a Throwable is intended to be created once and reused. Throwables that implement this interface kept their state in a local data structure in the owning schedulable object. This means that data is only valid until the next StaticThrowable is thrown in the that schedulable object. Having a marker interface makes it easier to provide checking tools to ensure the proper throw sequence for all Throwables thrown from application code.

Methods

```
@SCJMayAllocate({})
@SCJAllowed
@SCJMaySelfSuspend(false)
@SCJPhase({
    javax.safetycritical.annotate.Phase.STARTUP,
    javax.safetycritical.annotate.Phase.INITIALIZATION,
    javax.safetycritical.annotate.Phase.RUN,
    javax.safetycritical.annotate.Phase.CLEANUP })
public java.lang.Throwable fillInStackTrace( )
```

Calls the infrastructure to capture the current stack trace in the schedulable object's local memory.

returns a reference to this Throwable.

```
@SCJMayAllocate({})
@SCJAllowed
@SCJMaySelfSuspend(false)
@SCJPhase({
    javax.safetycritical.annotate.Phase.STARTUP,
    javax.safetycritical.annotate.Phase.INITIALIZATION,
    javax.safetycritical.annotate.Phase.RUN,
    javax.safetycritical.annotate.Phase.CLEANUP })
public java.lang.Throwable getCause( )
```

getCause returns the cause of this exception or null when no cause was set by initCause. The cause is another exception that was caught just before this exception was thrown.

returns The cause or null.

```
@SCJMayAllocate({})
@SCJAllowed
@SCJMaySelfSuspend(false)
@SCJPhase({
   javax.safetycritical.annotate.Phase.STARTUP,
   javax.safetycritical.annotate.Phase.INITIALIZATION,
   javax.safetycritical.annotate.Phase.RUN,
   javax.safetycritical.annotate.Phase.CLEANUP })
public java.lang.String getLocalizedMessage( )
```

Subclasses may override this message to get an error message that is localized to the default locale.

By default it returns getMessage().

returns the error message

```
@SCJMayAllocate({})
@SCJAllowed
@SCJMaySelfSuspend(false)
@SCJPhase({
   javax.safetycritical.annotate.Phase.STARTUP,
   javax.safetycritical.annotate.Phase.INITIALIZATION,
   javax.safetycritical.annotate.Phase.RUN,
   javax.safetycritical.annotate.Phase.CLEANUP })
public java.lang.String getMessage( )
```

get the message describing the problem from the schedulable object's local memory.

returns the message given to the constructor or null when no message was set.

```
@SCJMayAllocate({})
@SCJAllowed
@SCJMaySelfSuspend(false)
@SCJPhase({
   javax.safetycritical.annotate.Phase.STARTUP,
   javax.safetycritical.annotate.Phase.INITIALIZATION,
   javax.safetycritical.annotate.Phase.RUN,
   javax.safetycritical.annotate.Phase.CLEANUP })
public java.lang.StackTraceElement[] getStackTrace( )
```

Get the stack trace created by fillInStackTrace for this Throwable as an array of StackTraceElements.

The stack trace does not need to contain entries for all methods that are actually on the call stack, the infrastructure may decide to skip some stack trace entries. Even an empty array is a valid result of this function.

Repeated calls of this function without intervening calls to fillInStackTrace will return the same result.

When memory areas are used, and this Throwable was allocated in a different memory area than the current allocation context, the resulting stack trace will be allocated in either the same memory area this was allocated in or the current memory area, depending on which is the least deeply nested, thereby creating objects that are assignment compatible with both areas.

returns array representing the stack trace, it is never null.

```
@SCJMayAllocate({})
@SCJAllowed
@SCJMaySelfSuspend(false)
@SCJPhase({
    javax.safetycritical.annotate.Phase.STARTUP,
    javax.safetycritical.annotate.Phase.INITIALIZATION,
    javax.safetycritical.annotate.Phase.RUN,
    javax.safetycritical.annotate.Phase.CLEANUP })
public java.lang.Throwable initCause(Throwable causingThrowable)
```

Initializes the cause to the given Throwable in the schedulable object's local memory.

causingThrowable — the reason why this Throwable gets Thrown.

returns the reference to this Throwable.

Throws IllegalArgumentException when the cause is this Throwable itself.

```
@SCJMayAllocate({})
@SCJAllowed
@SCJMaySelfSuspend(false)
@SCJPhase({
    javax.safetycritical.annotate.Phase.STARTUP,
    javax.safetycritical.annotate.Phase.INITIALIZATION,
    javax.safetycritical.annotate.Phase.RUN,
    javax.safetycritical.annotate.Phase.CLEANUP })
public java.lang.Throwable initMessage(String message)
```

Set the message in the schedulable object's local storage. This is the only method that is not also defined in Throwable.

message — is the text to set.

```
@SCJMayAllocate({})
@SCJAllowed
@SCJMaySelfSuspend(false)
@SCJPhase({
    javax.safetycritical.annotate.Phase.STARTUP,
    javax.safetycritical.annotate.Phase.INITIALIZATION,
    javax.safetycritical.annotate.Phase.RUN,
    javax.safetycritical.annotate.Phase.CLEANUP })
public void printStackTrace(PrintStream stream)
```

Print the stack trace of this Throwable to the given stream.

The printed stack trace contains the result of toString() as the first line followed by one line for each stack trace element that contains the name of the method or constructor, optionally followed by the source file name and source file line number when available.

stream — the stream to print to.

```
@SCJMayAllocate({})
@SCJAllowed
@SCJMaySelfSuspend(false)
@SCJPhase({
    javax.safetycritical.annotate.Phase.STARTUP,
    javax.safetycritical.annotate.Phase.INITIALIZATION,
    javax.safetycritical.annotate.Phase.RUN,
    javax.safetycritical.annotate.Phase.CLEANUP })
public void printStackTrace( )
```

Print stack trace of this Throwable to System.err.

The printed stack trace contains the result of toString() as the first line followed by one line for each stack trace element that contains the name of the method or constructor, optionally followed by the source file name and source file line number when available.

```
@SCJMayAllocate({})
@SCJAllowed
@SCJMaySelfSuspend(false)
@SCJPhase({
    javax.safetycritical.annotate.Phase.STARTUP,
    javax.safetycritical.annotate.Phase.INITIALIZATION,
    javax.safetycritical.annotate.Phase.RUN,
    javax.safetycritical.annotate.Phase.CLEANUP })
public void setStackTrace(StackTraceElement [] new_stackTrace)
  throws java.lang.NullPointerException
```

This method allows overriding the stack trace that was filled during construction of this object. It is intended to be used in a serialization context when the stack trace of a remote exception should be treated like a local.

new_stackTrace — the stack trace to replace be used.

Throws NullPointerException when new_stackTrace or any element of new_stackTrace is null.

11.4.3 Class javax.realtime.StaticThrowableStorage

Declaration

```
@SCJAllowed
 public class StaticThrowableStorage implements javax.realtime.StaticThrowable
     extends java.lang.Throwable
```

Description

Provide the methods for managing the thread local memory used for storing the data needed by preallocated throwables, i.e., exceptions and errors which implement StaticThrowable. This call is visible so that an application can extend an existing conventional Java throwable and still implement StaticThrowable; its methods can be implemented using the methods defined in this class. An application defined throwable that does not need to extend an existing conventional Java throwable should extend on of StaticCheckedException, StaticRuntimeException, or StaticError instead.

Methods

```
@Override
@SCJMayAllocate({})
@SCJAllowed
@SCJMaySelfSuspend(false)
@SCJPhase({
   javax.safetycritical.annotate.Phase.STARTUP,
   javax.safetycritical.annotate.Phase.INITIALIZATION,
   javax.safetycritical.annotate.Phase.RUN,
   javax.safetycritical.annotate.Phase.CLEANUP })
public java.lang.Throwable fillInStackTrace( )
```

Capture the current thread's stack trace and save it in thread local storage. Only the part of the stack trace that fits in the preallocated buffer is stored. This method should be called by a preallocated exception to implement its method of the same name.

returns this

```
@Override
@SCJMayAllocate({})
@SCJAllowed
@SCJMaySelfSuspend(false)
@SCJPhase({
    javax.safetycritical.annotate.Phase.STARTUP,
    javax.safetycritical.annotate.Phase.INITIALIZATION,
    javax.safetycritical.annotate.Phase.RUN,
    javax.safetycritical.annotate.Phase.CLEANUP })
public java.lang.Throwable getCause( )
```

Get the cause from thread local storage that was saved by the last preallocated exception thrown. The actual exception that of the cause is not saved, but just a reference to its type. This returns a newly allocated exception without any valid content, i.e., no valid stack trace. This method should be called by a preallocated exception to implement its method of the same name.

returns the message

```
@SCJMayAllocate({})
@SCJAllowed
@SCJMaySelfSuspend(false)
@SCJPhase({
    javax.safetycritical.annotate.Phase.STARTUP,
    javax.safetycritical.annotate.Phase.INITIALIZATION,
    javax.safetycritical.annotate.Phase.RUN,
    javax.safetycritical.annotate.Phase.CLEANUP })
public static javax.realtime.StaticThrowableStorage getCurrent( )
```

A means of obtaining the storage object for the current task.

returns the storage object for the current task.

```
@Override
@SCJMayAllocate({})
@SCJAllowed
@SCJMaySelfSuspend(false)
@SCJPhase({
    javax.safetycritical.annotate.Phase.STARTUP,
    javax.safetycritical.annotate.Phase.INITIALIZATION,
    javax.safetycritical.annotate.Phase.RUN,
    javax.safetycritical.annotate.Phase.CLEANUP })
public java.lang.String getLocalizedMessage( )
```

```
@Override
@SCJMayAllocate({})
```

```
@SCJAllowed
@SCJMaySelfSuspend(false)
@SCJPhase({
   javax.safetycritical.annotate.Phase.STARTUP,
   javax.safetycritical.annotate.Phase.INITIALIZATION,
   javax.safetycritical.annotate.Phase.RUN,
   javax.safetycritical.annotate.Phase.CLEANUP })
public java.lang.String getMessage( )
```

Get the message from thread local storage that was saved by the last preallocated exception thrown. This method should be called by a preallocated exception to implement its method of the same name.

returns the message

```
@Override
@SCJMayAllocate({})
@SCJAllowed
@SCJMaySelfSuspend(false)
@SCJPhase({
   javax.safetycritical.annotate.Phase.STARTUP,
   javax.safetycritical.annotate.Phase.INITIALIZATION,
   javax.safetycritical.annotate.Phase.RUN,
   javax.safetycritical.annotate.Phase.CLEANUP })
public java.lang.StackTraceElement[] getStackTrace( )
```

Get the stack trace from thread local storage that was saved by the last preallocated exception thrown. This method should be called by a preallocated exception to implement its method of the same name.

returns an array of the elements of the stack trace.

```
@Override
@SCJMayAllocate({})
@SCJAllowed
@SCJMaySelfSuspend(false)
@SCJPhase({
   javax.safetycritical.annotate.Phase.STARTUP,
   javax.safetycritical.annotate.Phase.INITIALIZATION,
   javax.safetycritical.annotate.Phase.RUN,
   javax.safetycritical.annotate.Phase.CLEANUP })
public java.lang.Throwable initCause(Throwable causingThrowable)
```

Save the message in thread local storage for later retrieval. Only a reference to the exception class is stored. The rest of its information is lost. This method should be called by a preallocated exception to implement its method of the same name.

causingThrowable —

returns this

```
@Override
@SCJMayAllocate({})
@SCJAllowed
@SCJMaySelfSuspend(false)
@SCJPhase({
    javax.safetycritical.annotate.Phase.STARTUP,
    javax.safetycritical.annotate.Phase.INITIALIZATION,
    javax.safetycritical.annotate.Phase.RUN,
    javax.safetycritical.annotate.Phase.CLEANUP })
public java.lang.Throwable initMessage(String message)
```

Save the message in thread local storage for later retrieval. Only the part of the message that fits in the preallocated buffer is stored. This method should be called by a preallocated exception to implement its method of the same name.

message — the message to save.

```
@SCJMayAllocate({})
@SCJAllowed
@SCJMaySelfSuspend(false)
@Override
@SCJPhase({
    javax.safetycritical.annotate.Phase.STARTUP,
    javax.safetycritical.annotate.Phase.INITIALIZATION,
    javax.safetycritical.annotate.Phase.RUN,
    javax.safetycritical.annotate.Phase.CLEANUP })
public void printStackTrace(PrintStream stream)
```

```
@Override
@SCJMayAllocate({})
@SCJAllowed
@SCJMaySelfSuspend(false)
@SCJPhase({
    javax.safetycritical.annotate.Phase.STARTUP,
    javax.safetycritical.annotate.Phase.INITIALIZATION,
    javax.safetycritical.annotate.Phase.RUN,
    javax.safetycritical.annotate.Phase.CLEANUP })
public void printStackTrace( )
```

```
@Override
@SCJMayAllocate({})
@SCJAllowed
@SCJMaySelfSuspend(false)
```

```
@SCJPhase({
    javax.safetycritical.annotate.Phase.STARTUP,
    javax.safetycritical.annotate.Phase.INITIALIZATION,
    javax.safetycritical.annotate.Phase.RUN,
    javax.safetycritical.annotate.Phase.CLEANUP })
public void setStackTrace(StackTraceElement [] new_stackTrace)
```

11.4.4 Class java.lang.Exception

Declaration

```
@SCJAllowed
 public class Exception implements java.io.Serializable extends
     java.lang.Throwable
```

Description

The class Exception and its subclasses are a form of Throwable that indicates conditions that a reasonable application might want to catch.

Constructors

```
@SCJAllowed
@SCJMaySelfSuspend(false)
@SCJMayAllocate({})
@SCJPhase({
    javax.safetycritical.annotate.Phase.STARTUP,
    javax.safetycritical.annotate.Phase.INITIALIZATION,
    javax.safetycritical.annotate.Phase.RUN,
    javax.safetycritical.annotate.Phase.CLEANUP })
public Exception(String msg, Throwable cause)
```

Constructs an Exception object with an optional detail message and an optional cause. If cause is null, Services.captureStackBacktrace(this) is called to save the backtrace associated with the current thread. If cause is not null, Services.captureStackBacktrace(this) is not called to avoid overwriting the backtrace associated with the cause.

msg — the detail message for this Exception object.

cause — the cause of this exception .

```
@SCJAllowed
@SCJMaySelfSuspend(false)
@SCJMayAllocate({})
@SCJPhase({
    javax.safetycritical.annotate.Phase.STARTUP,
    javax.safetycritical.annotate.Phase.INITIALIZATION,
```

```
    javax.safetycritical.annotate.Phase.RUN,
    javax.safetycritical.annotate.Phase.CLEANUP })
public Exception( )
```

This constructor behaves the same as calling Exception(null, null).

```
@SCJAllowed
@SCJMaySelfSuspend(false)
@SCJMayAllocate({})
@SCJPhase({
    javax.safetycritical.annotate.Phase.STARTUP,
    javax.safetycritical.annotate.Phase.INITIALIZATION,
    javax.safetycritical.annotate.Phase.RUN,
    javax.safetycritical.annotate.Phase.CLEANUP })
public Exception(String msg)
```

This constructor behaves the same as calling Exception(msg, null).

```
@SCJAllowed
@SCJMaySelfSuspend(false)
@SCJMayAllocate({})
@SCJPhase({
    javax.safetycritical.annotate.Phase.STARTUP,
    javax.safetycritical.annotate.Phase.INITIALIZATION,
    javax.safetycritical.annotate.Phase.RUN,
    javax.safetycritical.annotate.Phase.CLEANUP })
public Exception(Throwable cause)
```

This constructor behaves the same as calling Exception(null, cause).

11.4.5 Class javax.realtime.StaticRuntimeException

Declaration

```
@SCJAllowed
 public abstract class StaticRuntimeException implements
     javax.realtime.StaticThrowable extends java.lang.RuntimeException
```

Methods

```
@SCJMayAllocate({})
@SCJAllowed
@SCJMaySelfSuspend(false)
@Override
@SCJPhase({
```

```
    javax.safetycritical.annotate.Phase.STARTUP,
    javax.safetycritical.annotate.Phase.INITIALIZATION,
    javax.safetycritical.annotate.Phase.RUN,
    javax.safetycritical.annotate.Phase.CLEANUP })
public java.lang.Throwable fillInStackTrace( )
```

Calls the infrastructure to capture the current stack trace in the schedulable object's local memory.

returns a reference to this Throwable.

```
@SCJMayAllocate({})
@SCJAllowed
@SCJMaySelfSuspend(false)
@SCJPhase({
    javax.safetycritical.annotate.Phase.STARTUP,
    javax.safetycritical.annotate.Phase.INITIALIZATION,
    javax.safetycritical.annotate.Phase.RUN,
    javax.safetycritical.annotate.Phase.CLEANUP })
public static javax.realtime.StaticRuntimeException get( )
```

Get the preallocated version of this Throwable. Allocation is done in memory that acts like **javax.realtime.ImmortalMemory** . The message and cause are cleared and the stack trace is filled out.

returns the preallocated exception

```
@SCJMayAllocate({})
@SCJAllowed
@SCJMaySelfSuspend(false)
@Override
@SCJPhase({
    javax.safetycritical.annotate.Phase.STARTUP,
    javax.safetycritical.annotate.Phase.INITIALIZATION,
    javax.safetycritical.annotate.Phase.RUN,
    javax.safetycritical.annotate.Phase.CLEANUP })
public java.lang.Throwable getCause( )
```

getCause returns the cause of this exception or null when no cause was set by initCause. The cause is another exception that was caught just before this exception was thrown.

returns The cause or null.

```
@SCJMayAllocate({})
@SCJAllowed
@SCJMaySelfSuspend(false)
@Override
@SCJPhase({
    javax.safetycritical.annotate.Phase.STARTUP,
    javax.safetycritical.annotate.Phase.INITIALIZATION,
    javax.safetycritical.annotate.Phase.RUN,
    javax.safetycritical.annotate.Phase.CLEANUP })
public java.lang.String getLocalizedMessage( )
```

Subclasses may override this message to get an error message that is localized to the default locale.

By default it returns getMessage().

returns the error message

```
@SCJMayAllocate({})
@SCJAllowed
@SCJMaySelfSuspend(false)
@Override
@SCJPhase({
    javax.safetycritical.annotate.Phase.STARTUP,
    javax.safetycritical.annotate.Phase.INITIALIZATION,
    javax.safetycritical.annotate.Phase.RUN,
    javax.safetycritical.annotate.Phase.CLEANUP })
public java.lang.String getMessage( )
```

get the message describing the problem from the schedulable object's local memory.

returns the message given to the constructor or null when no message was set.

```
@SCJMayAllocate({})
@SCJAllowed
@SCJMaySelfSuspend(false)
@Override
@SCJPhase({
    javax.safetycritical.annotate.Phase.STARTUP,
    javax.safetycritical.annotate.Phase.INITIALIZATION,
    javax.safetycritical.annotate.Phase.RUN,
    javax.safetycritical.annotate.Phase.CLEANUP })
public java.lang.StackTraceElement[] getStackTrace( )
```

Get the stack trace created by fillInStackTrace for this Throwable as an array of StackTraceElements.

The stack trace does not need to contain entries for all methods that are actually on the call stack, the infrastructure may decide to skip some stack trace entries. Even an empty array is a valid result of this function.

Repeated calls of this function without intervening calls to fillInStackTrace will return the same result.

When memory areas are used, and this Throwable was allocated in a different memory area than the current allocation context, the resulting stack trace will be allocated in either the same memory area this was allocated in or the current memory area, depending on which is the least deeply nested, thereby creating objects that are assignment compatible with both areas.

returns array representing the stack trace, it is never null.

```
@SCJMayAllocate({})
@SCJAllowed
@SCJMaySelfSuspend(false)
@Override
@SCJPhase({
    javax.safetycritical.annotate.Phase.STARTUP,
    javax.safetycritical.annotate.Phase.INITIALIZATION,
    javax.safetycritical.annotate.Phase.RUN,
    javax.safetycritical.annotate.Phase.CLEANUP })
public java.lang.Throwable initCause(Throwable causingThrowable)
```

Initializes the cause to the given Throwable in the schedulable object's local memory.

causingThrowable — the reason why this Throwable gets Thrown.

returns the reference to this Throwable.

Throws IllegalArgumentException when the cause is this Throwable itself.

```
@SCJMayAllocate({})
@SCJAllowed
@SCJMaySelfSuspend(false)
@Override
@SCJPhase({
    javax.safetycritical.annotate.Phase.STARTUP,
    javax.safetycritical.annotate.Phase.INITIALIZATION,
    javax.safetycritical.annotate.Phase.RUN,
    javax.safetycritical.annotate.Phase.CLEANUP })
public java.lang.Throwable initMessage(String message)
```

Set the message in the schedulable object's local storage. This is the only method that is not also defined in Throwable.

message — is the text to set.

```
@SCJMayAllocate({})
@SCJAllowed
@SCJMaySelfSuspend(false)
@Override
@SCJPhase({
    javax.safetycritical.annotate.Phase.STARTUP,
    javax.safetycritical.annotate.Phase.INITIALIZATION,
    javax.safetycritical.annotate.Phase.RUN,
    javax.safetycritical.annotate.Phase.CLEANUP })
public void printStackTrace(PrintStream stream)
```

Print stack trace of this Throwable to stream.

The printed stack trace contains the result of toString() as the first line followed by one line for each stack trace element that contains the name of the method or constructor, optionally followed by the source file name and source file line number when available.

```
@SCJMayAllocate({})
@SCJAllowed
@SCJMaySelfSuspend(false)
@Override
@SCJPhase({
    javax.safetycritical.annotate.Phase.STARTUP,
    javax.safetycritical.annotate.Phase.INITIALIZATION,
    javax.safetycritical.annotate.Phase.RUN,
    javax.safetycritical.annotate.Phase.CLEANUP })
public void printStackTrace( )
```

Print stack trace of this Throwable to System.err.

The printed stack trace contains the result of toString() as the first line followed by one line for each stack trace element that contains the name of the method or constructor, optionally followed by the source file name and source file line number when available.

```
@SCJMayAllocate({})
@SCJAllowed
@SCJMaySelfSuspend(false)
@Override
@SCJPhase({
    javax.safetycritical.annotate.Phase.STARTUP,
    javax.safetycritical.annotate.Phase.INITIALIZATION,
    javax.safetycritical.annotate.Phase.RUN,
```

```
    javax.safetycritical.annotate.Phase.CLEANUP })
public void setStackTrace(StackTraceElement [] new_stackTrace)
  throws java.lang.NullPointerException
```

This method allows overriding the stack trace that was filled during construction of this object. It is intended to be used in a serialization context when the stack trace of a remote exception should be treated like a local.

new_stackTrace — the stack trace to replace be used.

Throws NullPointerException when new_stackTrace or any element of new_stackTrace is null.

11.4.6 Class javax.realtime.StaticCheckedException

11.4.7 Class jaxax.realtime.ThrowBoundaryError

Declaration

```
@SCJAllowed
 public class ThrowBoundaryError implements java.io.Serializable extends
     javax.realtime.StaticError
```

Methods

```
@SCJMayAllocate({javax.safetycritical.annotate.AllocationContext.CURRENT})
@SCJAllowed
@SCJMaySelfSuspend(false)
@SCJPhase({
    javax.safetycritical.annotate.Phase.STARTUP,
    javax.safetycritical.annotate.Phase.INITIALIZATION,
    javax.safetycritical.annotate.Phase.RUN,
    javax.safetycritical.annotate.Phase.CLEANUP })
public static javax.realtime.ThrowBoundaryError get( )
```

Get the preallocated instance of this exception.

returns the preallocated instance of this exception.

11.4.8 Class java.lang.Error

Declaration

```
@SCJAllowed
 public class Error extends java.lang.Throwable
```

Description

An Error is a subclass of Throwable that indicates serious problems that a reasonable application should not try to catch.

Constructors

```
@SCJAllowed
@SCJMayAllocate({javax.safetycritical.annotate.AllocationContext.CURRENT})
@SCJMaySelfSuspend(false)
@SCJPhase({
   javax.safetycritical.annotate.Phase.STARTUP,
   javax.safetycritical.annotate.Phase.INITIALIZATION,
   javax.safetycritical.annotate.Phase.RUN,
   javax.safetycritical.annotate.Phase.CLEANUP })
public Error(String msg, Throwable cause)
```

Constructs an Error object with a specified detail message and with a specified cause. If cause is null, Services.captureStackBacktrace(this) is called to save the backtrace associated with the current thread. If cause is not null, Services.captureStackBacktrace(this) is not called to avoid overwriting the backtrace associated with the cause.

Allocates an application- and implementation-defined amount of memory in the current scope (to represent stack backtrace).

msg — the detail message for this Error object.

cause — the exception that caused this error.

```
@SCJAllowed
@SCJMayAllocate({javax.safetycritical.annotate.AllocationContext.CURRENT})
@SCJMaySelfSuspend(false)
@SCJPhase({
   javax.safetycritical.annotate.Phase.STARTUP,
   javax.safetycritical.annotate.Phase.INITIALIZATION,
   javax.safetycritical.annotate.Phase.RUN,
   javax.safetycritical.annotate.Phase.CLEANUP })
public Error( )
```

This constructor behaves the same as calling Error(String, Throwable) with the arguments (null, null).

```
@SCJAllowed
@SCJMayAllocate({javax.safetycritical.annotate.AllocationContext.CURRENT})
@SCJMaySelfSuspend(false)
@SCJPhase({
   javax.safetycritical.annotate.Phase.STARTUP,
```

```
    javax.safetycritical.annotate.Phase.INITIALIZATION,
    javax.safetycritical.annotate.Phase.RUN,
    javax.safetycritical.annotate.Phase.CLEANUP })
public Error(String msg)
```

This constructor behaves the same as calling Error(String, Throwable) with the arguments (msg, null).

```
@SCJAllowed
@SCJMayAllocate({javax.safetycritical.annotate.AllocationContext.CURRENT})
@SCJMaySelfSuspend(false)
@SCJPhase({
    javax.safetycritical.annotate.Phase.STARTUP,
    javax.safetycritical.annotate.Phase.INITIALIZATION,
    javax.safetycritical.annotate.Phase.RUN,
    javax.safetycritical.annotate.Phase.CLEANUP })
public Error(Throwable cause)
```

This constructor behaves the same as calling Error(String, Throwable) with the arguments (null, cause).

11.5 Rationale

SCJ allows individual managed schedulable objects to set aside different amounts of memory for backtrace information. During debugging, it is expected that developers may want to set aside large amounts of memory in order to maximize access to debugging information. However, during final deployment, many systems would run with minimal amounts of memory in order to reduce memory requirements and simplify the run-time behavior. Establishing the size of the stack backtrace buffer at ManagedSchedulable construction time relieves the SCJ implementation from having to dynamically allocate memory when dealing with throw boundary errors.

The support for stack traces is intended to enable the implementation to use a per-schedulable object reserved memory area of a predetermined size to hold the stack trace of the most recently caught exception.

One acceptable approach for an SCJ compliant implementations is the following:

- The constructor for java.lang.Throwable invokes StaticThrowable.FillInBackTrace() to save the current thread's stack backtrace into the thread-local buffer configured by this thread's ConfigurationParameters.

- StaticThrowable.FillInBackTrace() takes a single Throwable argument which is the object with which to associate the backtrace. captureBackTrace() saves a reference to its Throwable into a thread-local variable, using some run-time infrastructure mechanism if necessary, to avoid throwing an IllegalAssignmentError. At a subsequent invocation of Throwable.getStackTrace(), the run-time infrastructure code checks to make sure that the most recently captured stack backtrace information is associated with the Throwable being queried. If not, getStackTrace() returns a reference to a zero-length array which has been preallocated within immortal memory.
- Assuming that the current contents of the captured stack backtrace information is associated with the queried Throwable object, Throwable.getStackTrace() allocates and initializes an array of StackTraceElement, along with the StackTraceElement objects and the String objects referenced from the StackTraceElement objects, based on the current contents of the thread-local stack backtrace buffer.
- In case application programs desire to throw preallocated exceptions, the application program has the option to invoke StaticThrowable.FillInBackTrace() to overwrite the stack backtrace information associated with the previously allocated exception.
- The ThrowBoundaryError object that represents a thrown exception that crossed its scope boundary need not copy any information from the thread-local stack backtrace buffer at the time it replaces the thrown exception. When a thrown exception crosses its scope boundary, the thread-local ThrowBoundaryError object that is thrown in its place captures the class of the originally thrown exception and saves this as part of the ThrowBoundaryError object in support of the ThrowBoundaryError.getPropagatedMessage() method. Furthermore, the association for the thread-local stack backtrace buffer is changed from the Throwable that crossed its scope boundary to the ThrowBoundaryError.
- All of the exceptions thrown directly by the run-time infrastructure (such as ArithmeticException, OutOfMemoryError, StackOverflowError) are preallocated in immortal memory. Immediately before throwing a preallocated exception, the run-time infrastructure invokes StaticThrowable.FillInBackTrace() to overwrite the stack backtrace associated within the current schedulable with the preallocated exception.

SCJ defines its own ThrowBoundaryError class to stress that it works differently than the one in the RTSJ and to provide some additional methods. The ThrowBoundaryError exception is pre-allocated on a per-schedulable object basis; this ensures that its allocation upon detection of the boundary error cannot cause OutOfMemoryError to be thrown, that the exception is preserved even if scheduling occurs while it is being propagated, and that the exception cannot propagate out of its scope and thus cause a new ThrowBoundaryError exception to be thrown.

11.6 Compatibility

11.6.1 RTSJ Compatibility Issues

The precise semantics of ThrowBoundaryError differs from RTSJ to SCJ. In RTSJ, a new ThrowBoundaryError object is allocated in the enclosing memory area whenever the currently thrown exception crosses its scope boundary. In SCJ, the ThrowBoundaryError exception behaves as if it is pre-allocated on a per-schedulable object basis.

The SCJ allocation of ThrowBoundaryError in connection with a boundary error prevents secondary boundary error even if the exception is propagated through more scopes. Existing RTSJ code which is sensitive to the origin of ThrowBoundaryError would require changes to be used in an SCJ environment.

The SCJ limitation on the message length and stack trace size will require existing RTSJ code which algorithmically relies on the complete information to be changed to be used in an SCJ environment.

11.6.2 General Java Compatibility Issues

The SCJ restriction that the stack trace is only available for the most recently caught exception requires existing Java code which refers to older stack trace information to be changed to be used in an SCJ environment.

Chapter 12

Class Libraries for Safety-Critical Applications

For certifiable safety-critical systems, every library that the system uses must also be certifiable. Given the costs of the certification process, it is important to keep the size of every standard library as small as possible. Another consideration that argues for a smaller set of core libraries is the desire to reduce the need by application developers to subset the official standard for particular applications. In addition, many safety-critical software systems are missing features commonly used in other domains, such as file systems and networks. Therefore, the standard needs to accommodate both systems that require these features and those that do not.

SCJ is structured as a hierarchy of upwards compatible levels. Level 1 and Level 2 are designed to address the needs of systems that have more complexity and possibly more dynamic behavior than Level 0. Certain safety-critical library capabilities which are available to Level 2 programmers are not available to Level 1 and Level 0 programmers. Likewise, certain Level 1 libraries will not be available at Level 0.

Beyond the core libraries defined for the Level 0, Level 1, and Level 2 of SCJ, vendors may offer additional library support to complement the core capabilities.

See the javadoc appendices of this specification for descriptions of the required class libraries for safety-critical applications. Note that the SCJ annotations described in Chapter 9 are applied to all Java library classes and interfaces as well as all RTSJ classes and interfaces, just as they are also applied to the classes and interfaces defined in this specification.

The remainder of this chapter summarizes the minimally required capabilites of the four standard Java packages that shall be provided in an SCJ implementation. These descriptions refer to the libraries defined in JDK 1.8. Where differences in the capabilites required for SCJ exist, a brief discussion of the rationale for those differences is provided.

12.1 Minimal JDK 1.8 java.lang package Capabilities Required in SCJ Implementations

The `java.lang` package for SCJ ia almost exactly the same as in JDK 1.8 except for certain elements that are not required because of their complexity (with correspondingly high cost during safety certification) and their minimal usefulness in safety-critical applications. The following table describes the requirements for this package in SCJ implementations:

Table 12.1: java.lang Classes and Interfaces in SCJ

Class/Interface	Type	Discussion
`Appendable`	Interface	Same as JDK 1.8
`CharSequence`	Interface	Same as JDK 1.8
`Cloneable`	Interface	Not included in SCJ because it has been determined that it does not represent a reliable way to make deep copies within nested memory scopesSCJ
`Comparable`	Interface	Same as JDK 1.8
`Iterable`	Interface	Not included in SCJ to reduce size and complexity
`Readable`	Interface	Not included in SCJ to reduce size and complexity
`Runnable`	Interface	Same as JDK 1.8
`Thread.Uncaught-ExceptionHandler`	Interface	Same as JDK 1.8
`Boolean`	Class	Same as JDK 1.8
`Byte`	Class	Same as JDK 1.8
`Character`	Class	See section 12.1.1 for SCJ differences
`Class`	Class	See section 12.1.2 for SCJ differences
`ClassLoader`	Interface	Not included in SCJ to reduce size and complexity
`Compiler`	Interface	Not included in SCJ to reduce size and complexity
`Double`	Class	Same as JDK 1.8

Continued on next page

Table 12.1 – *Continued from previous page*

Class/Interface	Type	Discussion
`Enum`	Class	Same as JDK 1.8 except that SCJ does not require the `finalize()` or `valueof(Class<T> enumType, String name)` methods
`Float`	Class	Same as JDK 1.8
`InheritableThreadLocal`	Class	Not included in SCJ to reduce size and complexity
`Integer`	Class	Same as JDK 1.8
`Long`	Class	Same as JDK 1.8
`Math`	Class	Same as JDK 1.8
`Number`	Class	Same as JDK 1.8
`Object`	Class	See section 12.1.3 for SCJ differences
`Package`	Class	Not included in SCJ because reflection is severely limited to reduce size and complexity
`Process`	Class	Not included in SCJ because the services offered by this class will normally not be available with safety-certifiable operating environments
`ProcessBuilder`	Class	Not included in SCJ because the services offered by this class will normally not be available with safety-certifiable operating environments
`Runtime`	Class	Not included in SCJ because the services offered by this class will normally not be available with safety-certifiable operating environments and/or are not relevant in the absence of garbage collection and finalization

Continued on next page

Table 12.1 – *Continued from previous page*

Class/Interface	Type	Discussion
`RuntimePermission`	Class	Not included in SCJ because SCJ does not support on-the-fly security management – it is expected that safety-critical applications will assure security using static rather than dynamic techniques
`SecurityManager`	Class	Not included in SCJ because SCJ does not support on-the-fly security management – it is expected that safety-critical applications will assure security using static rather than dynamic techniques
`Short`	Class	Same as JDK 1.8
`StackTraceElement`	Class	Same as JDK 1.8
`StrictMath`	Class	Same as JDK 1.8
`String`	Class	See section 12.1.4 for SCJ differences
`StringBuffer`	Class	Not included in SCJ because SCJ assumes the use of JDK 1.8 or later which generates uses of `StringBuilder` instead of `StringBuffer`
`StringBuilder`	Class	See section 12.1.5 for SCJ differences
`System`	Class	See section 12.1.6 for SCJ differences
`Thread`	Class	See section 12.1.7 for SCJ differences
`ThreadGroup`	Class	Not included in SCJ to reduce size and complexity
`ThreadLocal`	Class	Not included in SCJ to reduce size and complexity
`Throwable`	Class	See section 12.1.8 for SCJ differences

12.1.1 Modifications to java.lang.Character

The class `Character` required for SCJ is the same as in JDK 1.8 except that the SCJ version does not require certain fields and methods that are omitted to reduce the

size and complexity of SCJ applications. It has been determined that safety-critical code would be generally unlikely to require significant amounts of text processing.

In addition, the classes `Character.Subset` and `Character.UnicodeBlock` is not required for SCJ implementations to reduce the size and complexity of the `java.lang` package.

The following `Character` fields are not required:

```
$DIRECTIONALITY_ARABIC_NUMBER$,
$DIRECTIONALITY_BOUNDARY_NEUTRAL$,
$DIRECTIONALITY_COMMON_NUMBER_SEPARATOR$,
$DIRECTIONALITY_EUROPEAN_NUMBER$,
$DIRECTIONALITY_EUROPEAN_NUMBER_SEPARATOR$,
$DIRECTIONALITY_LEFT_TO_RIGHT$,
$DIRECTIONALITY_LEFT_TO_RIGHT_EMBEDDING$,
$DIRECTIONALITY_LEFT_TO_RIGHT_OVERRIDE$,
$DIRECTIONALITY_NONSPACING_MARK$,
$DIRECTIONALITY_OTHER_NEUTRALS$,
$DIRECTIONALITY_PARAGRAPH_SEPARATOR$,
$DIRECTIONALITY_POP_DIRECTIONAL_FORMAT$,
$DIRECTIONALITY_RIGHT_TO_LEFT$,
$DIRECTIONALITY_RIGHT_TO_LEFT_ARABIC$,
$DIRECTIONALITY_RIGHT_TO_LEFT_EMBEDDING$,
$DIRECTIONALITY_RIGHT_TO_LEFT_OVERRIDE$,
$DIRECTIONALITY_SEGMENT_SEPARATOR$,
$DIRECTIONALITY_UNDEFINED$,
$DIRECTIONALITY_WHITESPACE$,
$MAX_CODE_POINT$,
$MAX_HIGH_SURROGATE$,
$MAX_LOW_SURROGATE$,
$MAX_SURROGATE$,
$MIN_CODE_POINT$,
$MIN_HIGH_SURROGATE$,
$MIN_LOW_SURROGATE$,
$MIN_SUPPLEMENTARY_CODE_POINT$,
$MIN_SURROGATE$
```

In addition, the following `Character` methods are not required:

```
charCount(int codePoint),
codePointAt(char[], int),
codePointAt(char[], int, int),
codePointAt(CharSequence, int),
codePointBefore(char[], int),
```

```
codePointBefore(char[], int, int),
codePointBefore(CharSequence, int),
codePointCount(char, int, int),
codePointCount(CharSequence, int, int),
digit(int codePoint, int),
forDigit(int, int),
getDirectionality(char),
getDirectionality(int),
getNumericValue(char),
getNumericValue(int),
getType(int codePoint),
isDefined(char),
isDefined(int),
isDigit(char),
isDigit(int),
isHighSurrogate(char),
isIdentifierIgnorable(char),
isIdentifierIgnorable(int codePoint),
isISOControl(char),
isISOControl(int codePoint),
isJavaIdentifierPart(char),
isJavaIdentifierPart(int),
isJavaIdentiferStart(char),
isJavaIdentifierStart(int codePoint),
isJavaLetter(char),
isJavaLetterOrDigit(char),
isLetter(int codePoint),
isLetterOrDigit(int codePoint),
isLowerCase(int codePoint),
isLowSurrogate(char),
isMirrored(char),
isMirrored(int codePoint),
isSpace(char),
isSupplementaryCodePoint(int codePoint),
isSurrogatePair(char, char),
isTitleCase(char),
isTitleCase(int codePoint),
isUnicodeIdentifierPart(char),
isUnicodeIdentifierStart(char),
isUnicodeIdentifierStart(int codePoint),
isUpperCase(int codePoint),
isWhitespace(int codePoint),
```

```
offsetByCodePoints(char[], int, int, int, int),
offsetByCodePoints(CharSequence, int, int),
reverseBytes(char),
toChars(int codePoint),
toChars(int codePoint, char[] int),
toCodePoint(char, char),
toLowerCase(int),
toTitleCase(char),
toTitleCase(int codePoint),
toUpperCase(int codePoint)
```

12.1.2 Modifications to java.lang.Class

The class `Class` required for SCJ is the same as in JDK 1.8 except that the SCJ version does not require certain interfaces and methods that are omitted to reduce the size and complexity of SCJ applications. Also, it has been decided that SCJ should severely restrict reflection to reduce SCJ complexity. Therefore the following interfaces are not required:

`AnnotatedElement`,
`GenericDeclaration`, or
`Type`.

In addition, the SCJ specification does not require the following methods:

```
asSubClass(Class),
cast(Object),
forName(String),
forName(String, boolean, ClassLoader),
getAnnotation(Class), getAnnotations(),
getCanonicalName(),
getClasses(),
getClassLoader(),
getConstructor(Class ...),
getConstructors(),
getDeclaredAnnotations(),
getDeclaredClasses(),
getDeclaredConstructor(Class ...),
getDeclaredConstructors(),
getDeclaredField(String),
getDeclaredFields(),
getDeclaredMethod(String, Class ...),
getDeclaredMethods(),
```

```
getEnclosingClass(),
getEnclosingConstructor(),
getEnclosingMethod(),
getFields(),
getGenericInterfaces(),
getGenericSuperclass(),
getInterfaces(),
getMethod(String, Class, ...),
getMethods(),
getModifiers(),
getPackage(),
getProtectionDomain(),
getResource(String),
getResourceAsStream(String),
getSigners(),
getSimpleName(),
getTypeParameters(),
isAnnotationPresent(),
isAnonymousClass(),
isLocalClass(),
isMemberClass(),
isPrimitive(),
isSynthetic(),
newInstance().
```

Note that the `Class` class does not require implementation of the following methods:

```
getEnumConstants(),
getSuperclass(),
isAnnotation(),
isArray(),
isAssignableFrom(Class),
isEnum(),
isInstance(Object),
isInterface(),
newInstance(),
toString().
```

12.1.3 Modifications to java.lang.Object

The class `Object` required for SCJ is the same as in JDK 1.8 except that the SCJ version does not require certain methods that are omitted to reduce the size and complexity of SCJ applications.

In SCJ the `finalize()` method is not `@SCJAllowed`. This means safety-critical applications shall not override this method.

The `clone()` method is not `@SCJAllowed` as its default shallow-copy behavior is not compatible with SCJ scoped memory usage patterns.

The following `Object` methods are `@SCJAllowed` only for Level 2 applications to enable a simpler run-time environment and easier analysis of real-time schedulability in Level 0 and Level 1:

`notify()`,
`notifyAll()`,
`wait()`,
`wait(long timeout)`,
and `wait(long timeout, int nanos)`

12.1.4 Modifications to java.lang.String

The class `String` required for SCJ is the same as that in JDK 1.8 except that the SCJ version does not require certain constructors and methods that are omitted to reduce the size and complexity of SCJ applications. It has been determined that safety-critical programs will not do extensive text processing.

These constructors are not required:

`String(byte[], Charset)`,
`String(byte[], int)`, `String(byte[], int, int, CharSet)`,
`String(byte[], int, int, int)`,
`String(byte[], int, int, String)`,
`String(byte[], String)`, `String(int[], int, int)`,
`String(StringBuffer)` constructors.

In addition, these methods are not required:

`codePointAt(int)`,
`codePointBefore(int)`,
`codePointCount(int beginIndex, int endIndex)`,
`contentEquals(StringBuffer sb)`, `copyValueOf(char[])`,

```
copyValueOf(char[], int, int),
format(Locale, String, Object...  args),
format(String, Object...  args),
getBytes(Charset),
getBytes(int, int, byte[], int),
getBytes(String),
intern(),
matches(String regex),
offsetByCodePoints(int, int),
replaceAll(String regex, String replacement),
replaceFirst(String regex, String replacement),
split(String regex), split(String regex, int limit),
toLowerCase(Locale),
toUpperCase(Locale)
```

The field `$CASE_INSENSITIVE_ORDER$` is not required.

12.1.5 Modifications to java.lang.StringBuilder

The class `StringBuilder` required for SCJ is the same as that in JDK 1.8 except that the SCJ version does not require certain methods that are omitted to reduce the size and complexity of SCJ applications and to enable safe sharing of a `StringBuilder`'s backing character array with any Strings constructed from this `StringBuilder`. It has been determined that safety-critical programs will not do extensive text processing.

The following methods are not required:

```
append(StringBuffer),
appendCodePoint(int),
codePointAt(int),
codePointBefore(int),
codePointCount(int, int),
delete(int, int),
deleteCharAt(int),
insert(int, boolean),
insert(int, char),
insert(int, char[]),
insert(int, char[], int, int),
insert(int, CharSequence),
insert(int, CharSequence, int, int),
insert(int, double),
```

```
insert(int, float),
insert(int, int),
insert(int, long),
insert(int, obj),
offsetByCodePoints(int, int),
replace(int, int, String),
reverse(),
setCharAt(int, char),
trimToSize()
```

12.1.6 Modifications to java.lang.System

The class `System` required for SCJ is the same as that in JDK 1.8 except that the SCJ version does not require certain methods that are omitted to reduce the size and complexity of SCJ applications. Note that SCJ does not support garbage collection, security management, or file I/O.

The following fields are not required:

`err`,
`in`,
or `out`

In addition, the following methods are not required:

```
clearProperty(),
console(),
gc(),
getenv(),
getenv(String name),
getProperties(),
getSecurityManager(),
inheritedChannel(),
load(String),
loadLibrary(String),
mapLibraryName(String),
runFinalization(),
runFinalizersOnExit(booleean),
setErr(PrintStream),
setIn(InputStream),
setOut(PrintStream),
setProperties(Properties),
```

```
setProperty(String, String),
setSecurityManager(SecurityManager)
```

12.1.7 Modifications to java.lang.Thread

The class `Thread` required for SCJ is the same as that in JDK 1.8 except that the SCJ version does not require certain constructors, methods, and fields that are omitted to reduce the size and complexity of SCJ applications. Note that SCJ does not allow instantiation of `Threads` because it allows only execution of `NoHeapRealtimeThreads`.

The class `Thread.State internal` is not required for SCJ implementations. The `Thread.UncaughtExceptionHandler` interface required for SCJ is the same as in JDK 1.8.

The following fields are not required:

```
$MAX_PRIORITY$,
$MIN_PRIORITY$,
$NORM_PRIORITY$.
```

None of the constructors are `@SCJAllowed`.

The following methods are not required in SCJ implementations:

```
activeCount(),
checkAccess(),
countStackFrames(),
destroy(),
dumpStack(),
enumerate(Thread[]),
getAllStackTraces(),
getContextClassLoader(),
getId(),
getPriority(),
getStackTrace(),
getState(),
getThreadGroup(),
holdsLock(Object),
resume(),
setContextClassLoader(ClassLoader),
setDaemon(boolean),
setName(String),
setPriority(int),
```

`stop(Throwable)`,
`suspend()`

12.1.8 Modifications to java.lang.Throwable

The class `Throwable` required for SCJ is the same as that in JDK 1.8 except that the SCJ version does not require certain classes and methods that are omitted to reduce the size and complexity of SCJ applications. The following table describes the SCJ requirements:

Table 12.2: java.lang.Throwable Classes and Methods in SCJ

Class/Method	Type	Discussion
`ArithmeticException`	Class	Same as JDK 1.8
`ArrayIndexOutOf-BoundsException`	Class	Same as JDK 1.8
`ArrayStoreException`	Class	Same as JDK 1.8
`ClassCastException`	Class	Same as JDK 1.8
`ClassNotFoundException`	Class	Same as JDK 1.8
`CloneNot-Supported-Exception`	Class	Same as JDK 1.8
`Exception`	Class	Same as JDK 1.8
`IllegalArgument-Exception`	Class	Same as JDK 1.8
`IllegalMonitor-StateException`	Class	Same as JDK 1.8 except it is allowed only in Level 2
`IllegalStateException`	Class	Same as JDK 1.8
`IndexOutOfBounds-Exception`	Class	Same as JDK 1.8
`Instantiation-Exception`	Class	Same as JDK 1.8
`InterruptedException`	Class	Same as JDK 1.8
`NegativeArray-SizeException`	Class	Same as JDK 1.8
`NullPointerException`	Class	Same as JDK 1.8
`NumberFormatException`	Class	Same as JDK 1.8
`RuntimeException`	Class	Same as JDK 1.8
`IllegalAccessException`	Class	Not included in SCJ to reduce size and complexity
`EnumConstantNot-PresentException`	Class	Not included in SCJ to reduce size and complexity

Continued on next page

Table 12.2 – *Continued from previous page*

Class/Method	Type	Discussion
`fillInStackTrace`	Method	Not included in SCJ to reduce size and complexity
`getLocalizedMessage`	Method	Not included in SCJ to reduce size and complexity
`initCause(Throwable)`	Method	Not included in SCJ to reduce size and complexity
`printStackTrace()`	Method	Not included in SCJ to reduce size and complexity
`printStackTrace(PrintStream)`	Method	Not included in SCJ to reduce size and complexity
`printStackTrace(PrintWriter)`	Method	Not included in SCJ to reduce size and complexity
`setStackTrace`	Method	Not included in SCJ to reduce size and complexity
`toString`	Method	Not included in SCJ to reduce size and complexity – note that `Throwable` inherits a simple `toString()` method from `Object`
`StringIndexOutOfBoundsException`	Class	Same as JDK 1.8
`UnsupportedOperationException`	Class	Same as JDK 1.8
`AssertionError`	Class	Same as JDK 1.8
`Error`	Class	Same as JDK 1.8
`IncompatibleClassChangeError`	Class	Same as JDK 1.8
`InternalError`	Class	Same as JDK 1.8
`OutOfMemoryError`	Class	Same as JDK 1.8
`StackOverflowError`	Class	Same as JDK 1.8
`UnsatisfiedLinkError`	Class	Same as JDK 1.8
`VirtualMachineError`	Class	Same as JDK 1.8
`NoSuchFieldException`	Class	Not included in SCJ because SCJ does not support dynamic class loading
`NoSuchMethodException`	Class	Not included in SCJ because SCJ does not support dynamic class loading

Continued on next page

Table 12.2 – *Continued from previous page*

Class/Method	Type	Discussion
`SecurityException`	Class	Not included in SCJ because SCJ does not support dynamic security management
`TypeNotPresent-Exception`	Class	Not included in SCJ because SCJ does not support reflection
`AbstractMethodError`	Class	Not included in SCJ because it can only arise during dynamic class loading
`ClassCircularityError`	Class	Not included in SCJ because it can only arise during dynamic class loading
`ClassFormatError`	Class	Not included in SCJ because it can only arise during dynamic class loading
`ExceptionIn-InitializerError`	Class	Not included in SCJ to reduce size and complexity
`IllegalAccessError`	Class	Not included in SCJ because it can only arise during dynamic class loading
`InstantiationError`	Class	Not included in SCJ because it can only arise during dynamic class loading
`LinkageError`	Class	Not included in SCJ because it can only arise during dynamic class loading
`NoClassDefFoundError`	Class	Not included in SCJ because it can only arise during dynamic class loading
`NoSuchFieldError`	Class	Not included in SCJ because it can only arise during dynamic class loading
`NoSuchMethodError`	Class	Not included in SCJ because it can only arise during dynamic class loading
`ThreadDeath`	Class	Not included in SCJ because SCJ does not support the `Thread.stop()` method
`UnknownError`	Class	Not included in SCJ to reduce size and complexity

Continued on next page

Table 12.2 – *Continued from previous page*

Class/Method	Type	Discussion
`UnsupportedClass-VersionError`	Class	Not included in SCJ because it can only arise during dynamic class loading
`VerifyError`	Class	Not included in SCJ because it can only arise during dynamic class loading

The class `Deprecated`: SCJ specification is the same as JDK 1.8.

The class `Override`: SCJ specification is the same as JDK 1.8.

The class `SuppressWarnings`: SCJ specification is the same as JDK 1.8.

12.2 Minimal JDK 1.8 java.lang.annotation Capabilities Required in SCJ Implementations

The interface `Annotation`: SCJ specification is the same as JDK 1.8.

The enum `ElementType`: SCJ defines the same constants as JDK 1.8. (Ordinal values associated with enumerated constants may not be the same, unless we make an effort to assure they are identical.) SCJ does not define the `values()` or `valueOf()` methods, as their main use deals with dynamic processing of annotations, whereas the use of annotations within SCJ is intended to be static.

The enum `RetentionPolicy`: SCJ defines the same constants as JDK 1.8. (Ordinal values associated with enumerated constants may not be the same, unless we make an effort to assure they are identical.) SCJ does not define the `values()` or `valueOf()` methods, as their main use deals with dynamic processing of annotations, whereas the use of annotations within SCJ is intended to be static.

The class `AnnotationTypeMismatchException`: is omitted from SCJ specification because this exception is only thrown during dynamic processing of annotations, whereas the use of annotations within SCJ is intended to be static.

The class `IncompleteAnnotationException`: is omitted from SCJ specification because this exception is only thrown during dynamic processing of annotations, whereas the use of annotations within SCJ is intended to be static.

The class `AnnotationFormatError`: is omitted from SCJ specification because this exception is only thrown during dynamic class loading, whereas SCJ does not support dynamic class loading.

The class `Documented`: SCJ specification is the same as JDK 1.8.

The class `Inherited`: SCJ specification is the same as JDK 1.8.

The class `Retention`: SCJ specification is the same as JDK 1.8.

The class `Target`: SCJ specification is the same as JDK 1.8.

12.3 Minimal JDK 1.8 java.io Capabilities Required in SCJ Implementations

Within the `java.io package`, the only definitions provided by the SCJ specification are the Serializable interface and those interfaces needed to support the SCJ Input and Output classes defined in Chapter 6.1. The supported interfaces and classes are the same as those defined in JDK 1.8. See Appendix B for details.

SCJ includes the Serializable interface for compatibility with standard edition Java. However, SCJ does not include any services to perform serialization, because such services would add undesirable size and complexity. For the same reason, SCJ omits other java.io services such as file access and formatted output.

12.4 Minimal JDK 1.8 java.util Capabilities Required in SCJ Implementations

Within the `java.util package`, the only definition provided by the SCJ specification is the `Iterator` interface. This interface is the same as JDK 1.8.

Appendix A

Javadoc Description of Package java.io

Package Contents *Page*

A.1 Classes

A.2 Interfaces

A.2.1 INTERFACE Closeable

```
@SCJAllowed
public interface Closeable
```

Unless specified to the contrary, see JDK for Java 8 documentation.

Methods

```
@SCJAllowed
@SCJMayAllocate({
   javax.safetycritical.annotate.AllocationContext.CURRENT,
   javax.safetycritical.annotate.AllocationContext.INNER,
   javax.safetycritical.annotate.AllocationContext.OUTER})
@SCJMaySelfSuspend(true)
@SCJPhase({
   javax.safetycritical.annotate.Phase.STARTUP,
   javax.safetycritical.annotate.Phase.INITIALIZATION,
   javax.safetycritical.annotate.Phase.RUN,
   javax.safetycritical.annotate.Phase.CLEANUP })
public void close( )
```

Closes this stream and releases any system resources associated with it. If the stream is already closed then invoking this method has no effect.

Throws IOException

A.2.2 INTERFACE DataInput

```
@SCJAllowed
public interface DataInput
```

The DataInput interface provides for reading bytes from a binary stream and reconstructing from them data in any of the Java primitive types. There is also a facility for reconstructing a String from data in modified UTF-8 format. It is generally true of all the reading routines in this interface that if end of file is reached before the desired number of bytes has been read, an EOFException (which is a kind of IOException) is thrown. If any byte cannot be read for any reason other than end of file, an IOException other than EOFException is thrown. In particular, an IOException may be thrown if the input stream has been closed.

Methods

```
@SCJAllowed
@SCJMayAllocate({
   javax.safetycritical.annotate.AllocationContext.CURRENT,
   javax.safetycritical.annotate.AllocationContext.INNER,
   javax.safetycritical.annotate.AllocationContext.OUTER})
@SCJMaySelfSuspend(true)
@SCJPhase({
   javax.safetycritical.annotate.Phase.STARTUP,
   javax.safetycritical.annotate.Phase.INITIALIZATION,
   javax.safetycritical.annotate.Phase.RUN,
   javax.safetycritical.annotate.Phase.CLEANUP })
public boolean readBoolean( )
```

Reads one input byte and returns true if that byte is nonzero, false if that byte is zero. This method is suitable for reading the byte written by the writeBoolean method of interface DataOutput.

Throws IOException

```
@SCJAllowed
@SCJMayAllocate({
   javax.safetycritical.annotate.AllocationContext.CURRENT,
   javax.safetycritical.annotate.AllocationContext.INNER,
   javax.safetycritical.annotate.AllocationContext.OUTER})
@SCJMaySelfSuspend(true)
@SCJPhase({
   javax.safetycritical.annotate.Phase.STARTUP,
   javax.safetycritical.annotate.Phase.INITIALIZATION,
   javax.safetycritical.annotate.Phase.RUN,
   javax.safetycritical.annotate.Phase.CLEANUP })
public byte readByte( )
```

Reads and returns one input byte. The byte is treated as a signed value in the range -128 through 127, inclusive. This method is suitable for reading the byte written by the writeByte method of interface DataOutput.

Throws IOException

```
@SCJAllowed
@SCJMayAllocate({
   javax.safetycritical.annotate.AllocationContext.CURRENT,
   javax.safetycritical.annotate.AllocationContext.INNER,
   javax.safetycritical.annotate.AllocationContext.OUTER})
@SCJMaySelfSuspend(true)
@SCJPhase({
```

```
    javax.safetycritical.annotate.Phase.STARTUP,
    javax.safetycritical.annotate.Phase.INITIALIZATION,
    javax.safetycritical.annotate.Phase.RUN,
    javax.safetycritical.annotate.Phase.CLEANUP })
public char readChar( )
```

Reads an input char and returns the char value. A Unicode char is made up of two bytes. Let a be the first byte read and b be the second byte. The value returned is: (char)((a <<8) | (b & 0xff)) This method is suitable for reading bytes written by the writeChar method of interface DataOutput.

Throws IOException

```
@SCJAllowed
@SCJMayAllocate({
    javax.safetycritical.annotate.AllocationContext.CURRENT,
    javax.safetycritical.annotate.AllocationContext.INNER,
    javax.safetycritical.annotate.AllocationContext.OUTER})
@SCJMaySelfSuspend(true)
@SCJPhase({
    javax.safetycritical.annotate.Phase.STARTUP,
    javax.safetycritical.annotate.Phase.INITIALIZATION,
    javax.safetycritical.annotate.Phase.RUN,
    javax.safetycritical.annotate.Phase.CLEANUP })
public double readDouble( )
```

Reads eight input bytes and returns a double value. It does this by first constructing a long value in exactly the manner of the readlong method, then converting this long value to a double in exactly the manner of the method Double.longBitsToDouble. This method is suitable for reading bytes written by the writeDouble method of interface DataOutput.

Throws IOException

```
@SCJAllowed
@SCJMayAllocate({
    javax.safetycritical.annotate.AllocationContext.CURRENT,
    javax.safetycritical.annotate.AllocationContext.INNER,
    javax.safetycritical.annotate.AllocationContext.OUTER})
@SCJMaySelfSuspend(true)
@SCJPhase({
    javax.safetycritical.annotate.Phase.STARTUP,
    javax.safetycritical.annotate.Phase.INITIALIZATION,
    javax.safetycritical.annotate.Phase.RUN,
    javax.safetycritical.annotate.Phase.CLEANUP })
public float readFloat( )
```

Reads four input bytes and returns a float value. It does this by first constructing an int value in exactly the manner of the readInt method, then converting this int value to a float in exactly the manner of the method Float.intBitsToFloat. This method is suitable for reading bytes written by the writeFloat method of interface DataOutput.

Throws IOException

```
@SCJAllowed
@SCJMayAllocate({
   javax.safetycritical.annotate.AllocationContext.CURRENT,
   javax.safetycritical.annotate.AllocationContext.INNER,
   javax.safetycritical.annotate.AllocationContext.OUTER})
@SCJMaySelfSuspend(true)
@SCJPhase({
   javax.safetycritical.annotate.Phase.STARTUP,
   javax.safetycritical.annotate.Phase.INITIALIZATION,
   javax.safetycritical.annotate.Phase.RUN,
   javax.safetycritical.annotate.Phase.CLEANUP })
public void readFully(byte [] b, int off, int len)
  throws java.io.IOException
```

Reads len bytes from an input stream. This method blocks until one of the following conditions occurs: . len bytes of input data are available, in which case a normal return is made. . End of file is detected, in which case an EOFException is thrown. . An I/O error occurs, in which case an IOException other than EOFException is thrown. If b is null, a NullPointerException is thrown. If off is negative, or len is negative, or off+len is greater than the length of the array b, then an IndexOutOfBoundsException is thrown. If len is zero, then no bytes are read. Otherwise, the first byte read is stored into element b[off], the next one into b[off+1], and so on. The number of bytes read is, at most, equal to len.

Throws IOException

```
@SCJAllowed
@SCJMayAllocate({
   javax.safetycritical.annotate.AllocationContext.CURRENT,
   javax.safetycritical.annotate.AllocationContext.INNER,
   javax.safetycritical.annotate.AllocationContext.OUTER})
@SCJMaySelfSuspend(true)
@SCJPhase({
   javax.safetycritical.annotate.Phase.STARTUP,
   javax.safetycritical.annotate.Phase.INITIALIZATION,
   javax.safetycritical.annotate.Phase.RUN,
   javax.safetycritical.annotate.Phase.CLEANUP })
```

public void readFully(**byte** [] b)
  throws java.io.IOException

Reads some bytes from an input stream and stores them into the buffer array b. The number of bytes read is equal to the length of b. This method blocks until one of the following conditions occurs: . b.length bytes of input data are available, in which case a normal return is made. . End of file is detected, in which case an EOFException is thrown. . An I/O error occurs, in which case an IOException other than EOFException is thrown. If b is null, a NullPointer-Exception is thrown. If b.length is zero, then no bytes are read. Otherwise, the first byte read is stored into element b[0], the next one into b[1], and so on. If an exception is thrown from this method, then it may be that some but not all bytes of b have been updated with data from the input stream.

Throws IOException

```
@SCJAllowed
@SCJMayAllocate({
   javax.safetycritical.annotate.AllocationContext.CURRENT,
   javax.safetycritical.annotate.AllocationContext.INNER,
   javax.safetycritical.annotate.AllocationContext.OUTER})
@SCJMaySelfSuspend(true)
@SCJPhase({
   javax.safetycritical.annotate.Phase.STARTUP,
   javax.safetycritical.annotate.Phase.INITIALIZATION,
   javax.safetycritical.annotate.Phase.RUN,
   javax.safetycritical.annotate.Phase.CLEANUP })
public int readInt( )
```

Reads four input bytes and returns an int value. Let a be the first byte read, b be the second byte, c be the third byte, and d be the fourth byte. The value returned is: (((a & 0xff) <<24) | ((b & 0xff) <<16) | ((c & 0xff) <<8) | (d & 0xff)) This method is suitable for reading bytes written by the writeInt method of interface DataOutput.

Throws IOException

```
@SCJAllowed
@SCJMayAllocate({
   javax.safetycritical.annotate.AllocationContext.CURRENT,
   javax.safetycritical.annotate.AllocationContext.INNER,
   javax.safetycritical.annotate.AllocationContext.OUTER})
@SCJMaySelfSuspend(true)
@SCJPhase({
   javax.safetycritical.annotate.Phase.STARTUP,
```

```
    javax.safetycritical.annotate.Phase.INITIALIZATION,
    javax.safetycritical.annotate.Phase.RUN,
    javax.safetycritical.annotate.Phase.CLEANUP })
public long readLong( )
```

Reads eight input bytes and returns a long value. Let a be the first byte read, b be the second byte, c be the third byte, d be the fourth byte, e be the fifth byte, f be the sixth byte, g be the seventh byte, and h be the eighth byte. The value returned is: (((long)(a & 0xff) <<56) | ((long)(b & 0xff) <<48) | ((long)(c & 0xff) <<40) | ((long)(d & 0xff) <<32) | ((long)(e & 0xff) <<24) | ((long)(f & 0xff) <<16) | ((long)(g & 0xff) <<8) | ((long)(h & 0xff))) This method is suitable for reading bytes written by the writeLong method of interface DataOutput.

Throws IOException

```
@SCJAllowed
@SCJMayAllocate({
    javax.safetycritical.annotate.AllocationContext.CURRENT,
    javax.safetycritical.annotate.AllocationContext.INNER,
    javax.safetycritical.annotate.AllocationContext.OUTER})
@SCJMaySelfSuspend(true)
@SCJPhase({
    javax.safetycritical.annotate.Phase.STARTUP,
    javax.safetycritical.annotate.Phase.INITIALIZATION,
    javax.safetycritical.annotate.Phase.RUN,
    javax.safetycritical.annotate.Phase.CLEANUP })
public short readShort( )
```

Reads two input bytes and returns a short value. Let a be the first byte read and b be the second byte. The value returned is: (short)((a <<8) | (b & 0xff)) This method is suitable for reading the bytes written by the writeShort method of interface DataOutput.

Throws IOException

```
@SCJAllowed
@SCJMayAllocate({
    javax.safetycritical.annotate.AllocationContext.CURRENT,
    javax.safetycritical.annotate.AllocationContext.INNER,
    javax.safetycritical.annotate.AllocationContext.OUTER})
@SCJMaySelfSuspend(true)
@SCJPhase({
    javax.safetycritical.annotate.Phase.STARTUP,
    javax.safetycritical.annotate.Phase.INITIALIZATION,
    javax.safetycritical.annotate.Phase.RUN,
    javax.safetycritical.annotate.Phase.CLEANUP })
public java.lang.String readUTF( )
```

Reads in a string that has been encoded using a modified UTF-8 format. The general contract of readUTF is that it reads a representation of a Unicode character string encoded in Java modified UTF-8 format; this string of characters is then returned as a String. First, two bytes are read and used to construct an unsigned 16-bit integer in exactly the manner of the readUnsignedShort method . This integer value is called the UTF length and specifies the number of additional bytes to be read. These bytes are then converted to characters by considering them in groups. The length of each group is computed from the value of the first byte of the group. The byte following a group, if any, is the first byte of the next group. If the first byte of a group matches the bit pattern 0xxxxxxx (where x means ”may be 0 or 1”), then the group consists of just that byte. The byte is zero-extended to form a character. If the first byte of a group matches the bit pattern 110xxxxx, then the group consists of that byte a and a second byte b. If there is no byte b (because byte a was the last of the bytes to be read), or if byte b does not match the bit pattern 10xxxxxx, then a UTFDataFormatException is thrown. Otherwise, the group is converted to the character: (char)(((a& 0x1F) <<6) | (b & 0x3F)) If the first byte of a group matches the bit pattern 1110xxxx, then the group consists of that byte a and two more bytes b and c. If there is no byte c (because byte a was one of the last two of the bytes to be read), or either byte b or byte c does not match the bit pattern 10xxxxxx, then a UTFDataFormatException is thrown. Otherwise, the group is converted to the character: (char)(((a & 0x0F) <<12) | ((b & 0x3F) <<6) | (c & 0x3F)) If the first byte of a group matches the pattern 1111xxxx or the pattern 10xxxxxx, then a UTFDataFormatException is thrown. If end of file is encountered at any time during this entire process, then an EOFException is thrown. After every group has been converted to a character by this process, the characters are gathered, in the same order in which their corresponding groups were read from the input stream, to form a String, which is returned. The writeUTF method of interface DataOutput may be used to write data that is suitable for reading by this method.

Throws IOException

Throws UTFDataFormatException

```
@SCJAllowed
@SCJMayAllocate({
   javax.safetycritical.annotate.AllocationContext.CURRENT,
   javax.safetycritical.annotate.AllocationContext.INNER,
   javax.safetycritical.annotate.AllocationContext.OUTER})
@SCJMaySelfSuspend(true)
@SCJPhase({
   javax.safetycritical.annotate.Phase.STARTUP,
   javax.safetycritical.annotate.Phase.INITIALIZATION,
```

```
    javax.safetycritical.annotate.Phase.RUN,
    javax.safetycritical.annotate.Phase.CLEANUP })
public int readUnsignedByte( )
```

Reads one input byte, zero-extends it to type int, and returns the result, which is therefore in the range 0 through 255. This method is suitable for reading the byte written by the writeByte method of interface DataOutput if the argument to writeByte was intended to be a value in the range 0 through 255.

Throws IOException

```
@SCJAllowed
@SCJMayAllocate({
    javax.safetycritical.annotate.AllocationContext.CURRENT,
    javax.safetycritical.annotate.AllocationContext.INNER,
    javax.safetycritical.annotate.AllocationContext.OUTER})
@SCJMaySelfSuspend(true)
@SCJPhase({
    javax.safetycritical.annotate.Phase.STARTUP,
    javax.safetycritical.annotate.Phase.INITIALIZATION,
    javax.safetycritical.annotate.Phase.RUN,
    javax.safetycritical.annotate.Phase.CLEANUP })
public int readUnsignedShort( )
```

Reads two input bytes, zero-extends it to type int, and returns an int value in the range 0 through 65535. Let a be the first byte read and b be the second byte. The value returned is: (((a & 0xff) <<8) | (b & 0xff)) This method is suitable for reading the bytes written by the writeShort method of interface DataOutput if the argument to writeShort was intended to be a value in the range 0 through 65535.

Throws IOException

```
@SCJAllowed
@SCJMayAllocate({
    javax.safetycritical.annotate.AllocationContext.CURRENT,
    javax.safetycritical.annotate.AllocationContext.INNER,
    javax.safetycritical.annotate.AllocationContext.OUTER})
@SCJMaySelfSuspend(true)
@SCJPhase({
    javax.safetycritical.annotate.Phase.STARTUP,
    javax.safetycritical.annotate.Phase.INITIALIZATION,
    javax.safetycritical.annotate.Phase.RUN,
    javax.safetycritical.annotate.Phase.CLEANUP })
public int skipBytes(int n)
  throws java.io.IOException
```

Makes an attempt to skip over n bytes of data from the input stream, discarding the skipped bytes. However, it may skip over some smaller number of bytes, possibly zero. This may result from any of a number of conditions; reaching end of file before n bytes have been skipped is only one possibility. This method never throws an EOFException. The actual number of bytes skipped is returned.

Throws IOException

A.2.3 INTERFACE DataOutput

```
@SCJAllowed
public interface DataOutput
```

The DataOutput interface provides for converting data from any of the Java primitive types to a series of bytes and writing these bytes to a binary stream. There is also a facility for converting a String into modified UTF-8 format and writing the resulting series of bytes. For all the methods in this interface that write bytes, it is generally true that if a byte cannot be written for any reason, an IOException is thrown.

Methods

```
@SCJAllowed
@SCJMayAllocate({
   javax.safetycritical.annotate.AllocationContext.CURRENT,
   javax.safetycritical.annotate.AllocationContext.INNER,
   javax.safetycritical.annotate.AllocationContext.OUTER})
@SCJMaySelfSuspend(true)
@SCJPhase({
   javax.safetycritical.annotate.Phase.STARTUP,
   javax.safetycritical.annotate.Phase.INITIALIZATION,
   javax.safetycritical.annotate.Phase.RUN,
   javax.safetycritical.annotate.Phase.CLEANUP })
public void write(int b)
  throws java.io.IOException
```

Writes the specified byte (the low eight bits of the argument b) to the underlying output stream.

Throws IOException

```
@SCJAllowed
@SCJMayAllocate({
```

```
    javax.safetycritical.annotate.AllocationContext.CURRENT,
    javax.safetycritical.annotate.AllocationContext.INNER,
    javax.safetycritical.annotate.AllocationContext.OUTER})
@SCJMaySelfSuspend(true)
@SCJPhase({
    javax.safetycritical.annotate.Phase.STARTUP,
    javax.safetycritical.annotate.Phase.INITIALIZATION,
    javax.safetycritical.annotate.Phase.RUN,
    javax.safetycritical.annotate.Phase.CLEANUP })
public void write(byte [] b, int off, int len)
  throws java.io.IOException
```

Writes len bytes from the specified byte array starting at offset off to the underlying output stream.

Throws IOException

```
@SCJAllowed
@SCJMayAllocate({
    javax.safetycritical.annotate.AllocationContext.CURRENT,
    javax.safetycritical.annotate.AllocationContext.INNER,
    javax.safetycritical.annotate.AllocationContext.OUTER})
@SCJMaySelfSuspend(true)
@SCJPhase({
    javax.safetycritical.annotate.Phase.STARTUP,
    javax.safetycritical.annotate.Phase.INITIALIZATION,
    javax.safetycritical.annotate.Phase.RUN,
    javax.safetycritical.annotate.Phase.CLEANUP })
public void writeBoolean(boolean v)
  throws java.io.IOException
```

Writes a boolean to the underlying output stream as a 1-byte value.

Throws IOException

```
@SCJAllowed
@SCJMayAllocate({
    javax.safetycritical.annotate.AllocationContext.CURRENT,
    javax.safetycritical.annotate.AllocationContext.INNER,
    javax.safetycritical.annotate.AllocationContext.OUTER})
@SCJMaySelfSuspend(true)
@SCJPhase({
    javax.safetycritical.annotate.Phase.STARTUP,
    javax.safetycritical.annotate.Phase.INITIALIZATION,
    javax.safetycritical.annotate.Phase.RUN,
    javax.safetycritical.annotate.Phase.CLEANUP })
public void writeByte(int v)
  throws java.io.IOException
```

Writes out a byte to the underlying output stream as a 1-byte value.

Throws IOException

```
@SCJAllowed
@SCJMayAllocate({
    javax.safetycritical.annotate.AllocationContext.CURRENT,
    javax.safetycritical.annotate.AllocationContext.INNER,
    javax.safetycritical.annotate.AllocationContext.OUTER})
@SCJMaySelfSuspend(true)
@SCJPhase({
    javax.safetycritical.annotate.Phase.STARTUP,
    javax.safetycritical.annotate.Phase.INITIALIZATION,
    javax.safetycritical.annotate.Phase.RUN,
    javax.safetycritical.annotate.Phase.CLEANUP })
public void writeChar(int v)
  throws java.io.IOException
```

Writes a char to the underlying output stream as a 2-byte value, high byte first.

Throws IOException

```
@SCJAllowed
@SCJMayAllocate({
    javax.safetycritical.annotate.AllocationContext.CURRENT,
    javax.safetycritical.annotate.AllocationContext.INNER,
    javax.safetycritical.annotate.AllocationContext.OUTER})
@SCJMaySelfSuspend(true)
@SCJPhase({
    javax.safetycritical.annotate.Phase.STARTUP,
    javax.safetycritical.annotate.Phase.INITIALIZATION,
    javax.safetycritical.annotate.Phase.RUN,
    javax.safetycritical.annotate.Phase.CLEANUP })
public void writeChars(String s)
  throws java.io.IOException
```

Writes a string to the underlying output stream as a sequence of characters.

Throws IOException

```
@SCJAllowed
@SCJMayAllocate({
    javax.safetycritical.annotate.AllocationContext.CURRENT,
    javax.safetycritical.annotate.AllocationContext.INNER,
    javax.safetycritical.annotate.AllocationContext.OUTER})
@SCJMaySelfSuspend(true)
@SCJPhase({
```

```
    javax.safetycritical.annotate.Phase.STARTUP,
    javax.safetycritical.annotate.Phase.INITIALIZATION,
    javax.safetycritical.annotate.Phase.RUN,
    javax.safetycritical.annotate.Phase.CLEANUP })
public void writeDouble(double v)
  throws java.io.IOException
```

Converts the double argument to a long using the doubleToLongBits method in class Double, and then writes that long value to the underlying output stream as an 8-byte quantity, high byte first.

Throws IOException

```
@SCJAllowed
@SCJMayAllocate({
    javax.safetycritical.annotate.AllocationContext.CURRENT,
    javax.safetycritical.annotate.AllocationContext.INNER,
    javax.safetycritical.annotate.AllocationContext.OUTER})
@SCJMaySelfSuspend(true)
@SCJPhase({
    javax.safetycritical.annotate.Phase.STARTUP,
    javax.safetycritical.annotate.Phase.INITIALIZATION,
    javax.safetycritical.annotate.Phase.RUN,
    javax.safetycritical.annotate.Phase.CLEANUP })
public void writeFloat(float v)
  throws java.io.IOException
```

Converts the float argument to an int using the floatToIntBits method in class Float, and then writes that int value to the underlying output stream as a 4-byte quantity, high byte first.

Throws IOException

```
@SCJAllowed
@SCJMayAllocate({
    javax.safetycritical.annotate.AllocationContext.CURRENT,
    javax.safetycritical.annotate.AllocationContext.INNER,
    javax.safetycritical.annotate.AllocationContext.OUTER})
@SCJMaySelfSuspend(true)
@SCJPhase({
    javax.safetycritical.annotate.Phase.STARTUP,
    javax.safetycritical.annotate.Phase.INITIALIZATION,
    javax.safetycritical.annotate.Phase.RUN,
    javax.safetycritical.annotate.Phase.CLEANUP })
public void writeInt(int v)
  throws java.io.IOException
```

Writes an int to the underlying output stream as four bytes, high byte first.

Throws IOException

```
@SCJAllowed
@SCJMayAllocate({
    javax.safetycritical.annotate.AllocationContext.CURRENT,
    javax.safetycritical.annotate.AllocationContext.INNER,
    javax.safetycritical.annotate.AllocationContext.OUTER})
@SCJMaySelfSuspend(true)
@SCJPhase({
    javax.safetycritical.annotate.Phase.STARTUP,
    javax.safetycritical.annotate.Phase.INITIALIZATION,
    javax.safetycritical.annotate.Phase.RUN,
    javax.safetycritical.annotate.Phase.CLEANUP })
public void writeLong(long v)
  throws java.io.IOException
```

Writes a long to the underlying output stream as eight bytes, high byte first.

Throws IOException

```
@SCJAllowed
@SCJMayAllocate({
    javax.safetycritical.annotate.AllocationContext.CURRENT,
    javax.safetycritical.annotate.AllocationContext.INNER,
    javax.safetycritical.annotate.AllocationContext.OUTER})
@SCJMaySelfSuspend(true)
@SCJPhase({
    javax.safetycritical.annotate.Phase.STARTUP,
    javax.safetycritical.annotate.Phase.INITIALIZATION,
    javax.safetycritical.annotate.Phase.RUN,
    javax.safetycritical.annotate.Phase.CLEANUP })
public void writeShort(int v)
  throws java.io.IOException
```

Writes a short to the underlying output stream as two bytes, high byte first.

Throws IOException

```
@SCJAllowed
@SCJMayAllocate({
    javax.safetycritical.annotate.AllocationContext.CURRENT,
    javax.safetycritical.annotate.AllocationContext.INNER,
    javax.safetycritical.annotate.AllocationContext.OUTER})
@SCJMaySelfSuspend(true)
@SCJPhase({
    javax.safetycritical.annotate.Phase.STARTUP,
    javax.safetycritical.annotate.Phase.INITIALIZATION,
    javax.safetycritical.annotate.Phase.RUN,
```

```
    javax.safetycritical.annotate.Phase.CLEANUP })
public void writeUTF(String str)
  throws java.io.IOException
```

Writes a string to the underlying output stream using UTF-8 encoding in a machine-independent manner.

Throws IOException

A.2.4 INTERFACE **Flushable**

```
@SCJAllowed
public interface Flushable
```

Unless specified to the contrary, see JDK for Java 8 documentation.

Methods

```
@SCJAllowed
@SCJMayAllocate({
    javax.safetycritical.annotate.AllocationContext.CURRENT,
    javax.safetycritical.annotate.AllocationContext.INNER,
    javax.safetycritical.annotate.AllocationContext.OUTER})
@SCJMaySelfSuspend(true)
@SCJPhase({
    javax.safetycritical.annotate.Phase.STARTUP,
    javax.safetycritical.annotate.Phase.INITIALIZATION,
    javax.safetycritical.annotate.Phase.RUN,
    javax.safetycritical.annotate.Phase.CLEANUP })
public void flush( )
```

Flushes this stream by writing any buffered output to the underlying stream.

Throws IOException

A.2.5 INTERFACE **Serializable**

```
@SCJAllowed
public interface Serializable
```

This interface is provided for compatibility with standard edition Java. However, JSR302 does not support serialization, so the presence or absence of this interface has no visible effect within a JSR302 application.

A.3 Classes

A.3.1 CLASS DataInputStream

```
@SCJAllowed
public class DataInputStream implements java.io.DataInput extends
    java.io.InputStream
```

A data input stream lets an application read primitive Java data types from an underlying input stream in a machine-independent way. An application uses a data output stream to write data that can later be read by a data input stream. DataInputStream is not necessarily safe for multithreaded access. Thread safety is optional and is the responsibility of users of methods in this class.

Fields

```
@SCJAllowed
protected java.io.InputStream in
```

The input stream.

Constructors

```
@SCJAllowed
@SCJMayAllocate({
   javax.safetycritical.annotate.AllocationContext.CURRENT,
   javax.safetycritical.annotate.AllocationContext.INNER,
   javax.safetycritical.annotate.AllocationContext.OUTER})
@SCJMaySelfSuspend(true)
@SCJPhase({
   javax.safetycritical.annotate.Phase.STARTUP,
   javax.safetycritical.annotate.Phase.INITIALIZATION,
   javax.safetycritical.annotate.Phase.RUN,
   javax.safetycritical.annotate.Phase.CLEANUP })
public DataInputStream(InputStream in)
```

Creates a DataInputStream and saves its argument, the input stream in, for later use.

Methods

```
@SCJAllowed
@SCJMayAllocate({
   javax.safetycritical.annotate.AllocationContext.CURRENT,
   javax.safetycritical.annotate.AllocationContext.INNER,
   javax.safetycritical.annotate.AllocationContext.OUTER})
@SCJMaySelfSuspend(true)
@SCJPhase({
   javax.safetycritical.annotate.Phase.STARTUP,
   javax.safetycritical.annotate.Phase.INITIALIZATION,
   javax.safetycritical.annotate.Phase.RUN,
   javax.safetycritical.annotate.Phase.CLEANUP })
public int available( )
```

Returns the number of bytes that can be read from this input stream without blocking. This method simply performs in.available() and returns the result.

Throws IOException

```
@SCJAllowed
@SCJMayAllocate({
   javax.safetycritical.annotate.AllocationContext.CURRENT,
   javax.safetycritical.annotate.AllocationContext.INNER,
   javax.safetycritical.annotate.AllocationContext.OUTER})
@SCJMaySelfSuspend(true)
@SCJPhase({
   javax.safetycritical.annotate.Phase.STARTUP,
   javax.safetycritical.annotate.Phase.INITIALIZATION,
   javax.safetycritical.annotate.Phase.RUN,
   javax.safetycritical.annotate.Phase.CLEANUP })
public void close( )
```

Closes this input stream and releases any system resources associated with the stream. This method simply performs in.close().

Throws IOException

```
@SCJAllowed
@SCJMayAllocate({
   javax.safetycritical.annotate.AllocationContext.CURRENT,
   javax.safetycritical.annotate.AllocationContext.INNER,
   javax.safetycritical.annotate.AllocationContext.OUTER})
@SCJMaySelfSuspend(true)
@SCJPhase({
   javax.safetycritical.annotate.Phase.STARTUP,
   javax.safetycritical.annotate.Phase.INITIALIZATION,
   javax.safetycritical.annotate.Phase.RUN,
   javax.safetycritical.annotate.Phase.CLEANUP })
public void mark(int readlimit)
```

Marks the current position in this input stream. A subsequent call to the reset method repositions this stream at the last marked position so that subsequent reads re-read the same bytes. The readlimit argument tells this input stream to allow that many bytes to be read before the mark position gets invalidated. This method simply performs in.mark(readlimit).

```
@SCJAllowed
@SCJMayAllocate({
   javax.safetycritical.annotate.AllocationContext.CURRENT,
   javax.safetycritical.annotate.AllocationContext.INNER,
   javax.safetycritical.annotate.AllocationContext.OUTER})
@SCJMaySelfSuspend(true)
@SCJPhase({
   javax.safetycritical.annotate.Phase.STARTUP,
   javax.safetycritical.annotate.Phase.INITIALIZATION,
   javax.safetycritical.annotate.Phase.RUN,
   javax.safetycritical.annotate.Phase.CLEANUP })
public boolean markSupported( )
```

Tests if this input stream supports the mark and reset methods. This method simply performs in.markSupported().

```
@SCJAllowed
@SCJMayAllocate({
   javax.safetycritical.annotate.AllocationContext.CURRENT,
   javax.safetycritical.annotate.AllocationContext.INNER,
   javax.safetycritical.annotate.AllocationContext.OUTER})
@SCJMaySelfSuspend(true)
@SCJPhase({
   javax.safetycritical.annotate.Phase.STARTUP,
   javax.safetycritical.annotate.Phase.INITIALIZATION,
   javax.safetycritical.annotate.Phase.RUN,
   javax.safetycritical.annotate.Phase.CLEANUP })
public final int read(byte [] b)
  throws java.io.IOException
```

See the general contract of the read method of DataInput. Bytes for this operation are read from the contained input stream.

Throws IOException

```
@SCJAllowed
@SCJMayAllocate({
   javax.safetycritical.annotate.AllocationContext.CURRENT,
   javax.safetycritical.annotate.AllocationContext.INNER,
```

```
    javax.safetycritical.annotate.AllocationContext.OUTER})
@SCJMaySelfSuspend(true)
@SCJPhase({
    javax.safetycritical.annotate.Phase.STARTUP,
    javax.safetycritical.annotate.Phase.INITIALIZATION,
    javax.safetycritical.annotate.Phase.RUN,
    javax.safetycritical.annotate.Phase.CLEANUP })
public final int read(byte [] b, int off, int len)
  throws java.io.IOException
```

Reads up to len bytes of data from this input stream into an array of bytes. This method blocks until some input is available. This method simply performs in.read(b, off, len) and returns the result.

Throws IOException

```
@SCJAllowed
@SCJMayAllocate({
    javax.safetycritical.annotate.AllocationContext.CURRENT,
    javax.safetycritical.annotate.AllocationContext.INNER,
    javax.safetycritical.annotate.AllocationContext.OUTER})
@SCJMaySelfSuspend(true)
@SCJPhase({
    javax.safetycritical.annotate.Phase.STARTUP,
    javax.safetycritical.annotate.Phase.INITIALIZATION,
    javax.safetycritical.annotate.Phase.RUN,
    javax.safetycritical.annotate.Phase.CLEANUP })
public int read( )
```

Reads the next byte of data from this input stream. The value byte is returned as an int in the range 0 to 255. If no byte is available because the end of the stream has been reached, the value -1 is returned. This method blocks until input data is available, the end of the stream is detected, or an exception is thrown. This method simply performs in.read() and returns the result.

```
@SCJAllowed
@SCJMayAllocate({
    javax.safetycritical.annotate.AllocationContext.CURRENT,
    javax.safetycritical.annotate.AllocationContext.INNER,
    javax.safetycritical.annotate.AllocationContext.OUTER})
@SCJMaySelfSuspend(true)
@SCJPhase({
    javax.safetycritical.annotate.Phase.STARTUP,
    javax.safetycritical.annotate.Phase.INITIALIZATION,
    javax.safetycritical.annotate.Phase.RUN,
    javax.safetycritical.annotate.Phase.CLEANUP })
public final boolean readBoolean( )
```

See the general contract of the readBoolean method of DataInput. Bytes for this operation are read from the contained input stream.

Throws IOException

```
@SCJAllowed
@SCJMayAllocate({
   javax.safetycritical.annotate.AllocationContext.CURRENT,
   javax.safetycritical.annotate.AllocationContext.INNER,
   javax.safetycritical.annotate.AllocationContext.OUTER})
@SCJMaySelfSuspend(true)
@SCJPhase({
   javax.safetycritical.annotate.Phase.STARTUP,
   javax.safetycritical.annotate.Phase.INITIALIZATION,
   javax.safetycritical.annotate.Phase.RUN,
   javax.safetycritical.annotate.Phase.CLEANUP })
public final byte readByte( )
```

See the general contract of the readByte method of DataInput. Bytes for this operation are read from the contained input stream.

Throws IOException

```
@SCJAllowed
@SCJMayAllocate({
   javax.safetycritical.annotate.AllocationContext.CURRENT,
   javax.safetycritical.annotate.AllocationContext.INNER,
   javax.safetycritical.annotate.AllocationContext.OUTER})
@SCJMaySelfSuspend(true)
@SCJPhase({
   javax.safetycritical.annotate.Phase.STARTUP,
   javax.safetycritical.annotate.Phase.INITIALIZATION,
   javax.safetycritical.annotate.Phase.RUN,
   javax.safetycritical.annotate.Phase.CLEANUP })
public final char readChar( )
```

See the general contract of the readChar method of DataInput. Bytes for this operation are read from the contained input stream.

Throws IOException

```
@SCJAllowed
@SCJMayAllocate({
   javax.safetycritical.annotate.AllocationContext.CURRENT,
   javax.safetycritical.annotate.AllocationContext.INNER,
   javax.safetycritical.annotate.AllocationContext.OUTER})
@SCJMaySelfSuspend(true)
```

```
@SCJPhase({
    javax.safetycritical.annotate.Phase.STARTUP,
    javax.safetycritical.annotate.Phase.INITIALIZATION,
    javax.safetycritical.annotate.Phase.RUN,
    javax.safetycritical.annotate.Phase.CLEANUP })
public final double readDouble( )
```

See the general contract of the readDouble method of DataInput. Bytes for this operation are read from the contained input stream.

Throws IOException

```
@SCJAllowed
@SCJMayAllocate({
    javax.safetycritical.annotate.AllocationContext.CURRENT,
    javax.safetycritical.annotate.AllocationContext.INNER,
    javax.safetycritical.annotate.AllocationContext.OUTER})
@SCJMaySelfSuspend(true)
@SCJPhase({
    javax.safetycritical.annotate.Phase.STARTUP,
    javax.safetycritical.annotate.Phase.INITIALIZATION,
    javax.safetycritical.annotate.Phase.RUN,
    javax.safetycritical.annotate.Phase.CLEANUP })
public final float readFloat( )
```

See the general contract of the readFloat method of DataInput. Bytes for this operation are read from the contained input stream.

Throws IOException

```
@SCJAllowed
@SCJMayAllocate({
    javax.safetycritical.annotate.AllocationContext.CURRENT,
    javax.safetycritical.annotate.AllocationContext.INNER,
    javax.safetycritical.annotate.AllocationContext.OUTER})
@SCJMaySelfSuspend(true)
@SCJPhase({
    javax.safetycritical.annotate.Phase.STARTUP,
    javax.safetycritical.annotate.Phase.INITIALIZATION,
    javax.safetycritical.annotate.Phase.RUN,
    javax.safetycritical.annotate.Phase.CLEANUP })
public final void readFully(byte [] b, int off, int len)
  throws java.io.IOException
```

See the general contract of the readFully method of DataInput. Bytes for this operation are read from the contained input stream.

Throws IOException

```
@SCJAllowed
@SCJMayAllocate({
   javax.safetycritical.annotate.AllocationContext.CURRENT,
   javax.safetycritical.annotate.AllocationContext.INNER,
   javax.safetycritical.annotate.AllocationContext.OUTER})
@SCJMaySelfSuspend(true)
@SCJPhase({
   javax.safetycritical.annotate.Phase.STARTUP,
   javax.safetycritical.annotate.Phase.INITIALIZATION,
   javax.safetycritical.annotate.Phase.RUN,
   javax.safetycritical.annotate.Phase.CLEANUP })
public final void readFully(byte [] b)
  throws java.io.IOException
```

See the general contract of the readFully method of DataInput. Bytes for this operation are read from the contained input stream.

Throws IOException

```
@SCJAllowed
@SCJMayAllocate({
   javax.safetycritical.annotate.AllocationContext.CURRENT,
   javax.safetycritical.annotate.AllocationContext.INNER,
   javax.safetycritical.annotate.AllocationContext.OUTER})
@SCJMaySelfSuspend(true)
@SCJPhase({
   javax.safetycritical.annotate.Phase.STARTUP,
   javax.safetycritical.annotate.Phase.INITIALIZATION,
   javax.safetycritical.annotate.Phase.RUN,
   javax.safetycritical.annotate.Phase.CLEANUP })
public final int readInt( )
```

See the general contract of the readInt method of DataInput. Bytes for this operation are read from the contained input stream.

Throws IOException

```
@SCJAllowed
@SCJMayAllocate({
   javax.safetycritical.annotate.AllocationContext.CURRENT,
   javax.safetycritical.annotate.AllocationContext.INNER,
   javax.safetycritical.annotate.AllocationContext.OUTER})
@SCJMaySelfSuspend(true)
@SCJPhase({
   javax.safetycritical.annotate.Phase.STARTUP,
```

```
    javax.safetycritical.annotate.Phase.INITIALIZATION,
    javax.safetycritical.annotate.Phase.RUN,
    javax.safetycritical.annotate.Phase.CLEANUP })
public final long readLong( )
```

See the general contract of the readLong method of DataInput. Bytes for this operation are read from the contained input stream.

Throws IOException

```
@SCJAllowed
@SCJMayAllocate({
    javax.safetycritical.annotate.AllocationContext.CURRENT,
    javax.safetycritical.annotate.AllocationContext.INNER,
    javax.safetycritical.annotate.AllocationContext.OUTER})
@SCJMaySelfSuspend(true)
@SCJPhase({
    javax.safetycritical.annotate.Phase.STARTUP,
    javax.safetycritical.annotate.Phase.INITIALIZATION,
    javax.safetycritical.annotate.Phase.RUN,
    javax.safetycritical.annotate.Phase.CLEANUP })
public final short readShort( )
```

See the general contract of the readShort method of DataInput. Bytes for this operation are read from the contained input stream.

Throws IOException

```
@SCJAllowed
@SCJMayAllocate({
    javax.safetycritical.annotate.AllocationContext.CURRENT,
    javax.safetycritical.annotate.AllocationContext.INNER,
    javax.safetycritical.annotate.AllocationContext.OUTER})
@SCJMaySelfSuspend(true)
@SCJPhase({
    javax.safetycritical.annotate.Phase.STARTUP,
    javax.safetycritical.annotate.Phase.INITIALIZATION,
    javax.safetycritical.annotate.Phase.RUN,
    javax.safetycritical.annotate.Phase.CLEANUP })
public final java.lang.String readUTF( )
```

See the general contract of the readUTF method of DataInput. Bytes for this operation are read from the contained input stream.

Throws IOException

```
@SCJAllowed
@SCJMayAllocate({
   javax.safetycritical.annotate.AllocationContext.CURRENT,
   javax.safetycritical.annotate.AllocationContext.INNER,
   javax.safetycritical.annotate.AllocationContext.OUTER})
@SCJMaySelfSuspend(true)
@SCJPhase({
   javax.safetycritical.annotate.Phase.STARTUP,
   javax.safetycritical.annotate.Phase.INITIALIZATION,
   javax.safetycritical.annotate.Phase.RUN,
   javax.safetycritical.annotate.Phase.CLEANUP })
public static final java.lang.String readUTF(DataInput in)
  throws java.io.IOException
```

Reads from the stream in a representation of a Unicode character string encoded in Java modified UTF-8 format; this string of characters is then returned as a String. The details of the modified UTF-8 representation are exactly the same as for the readUTF method of DataInput

Throws IOException

```
@SCJAllowed
@SCJMayAllocate({
   javax.safetycritical.annotate.AllocationContext.CURRENT,
   javax.safetycritical.annotate.AllocationContext.INNER,
   javax.safetycritical.annotate.AllocationContext.OUTER})
@SCJMaySelfSuspend(true)
@SCJPhase({
   javax.safetycritical.annotate.Phase.STARTUP,
   javax.safetycritical.annotate.Phase.INITIALIZATION,
   javax.safetycritical.annotate.Phase.RUN,
   javax.safetycritical.annotate.Phase.CLEANUP })
public final int readUnsignedByte( )
```

See the general contract of the readUnsignedByte method of DataInput. Bytes for this operation are read from the contained input stream.

Throws IOException

```
@SCJAllowed
@SCJMayAllocate({
   javax.safetycritical.annotate.AllocationContext.CURRENT,
   javax.safetycritical.annotate.AllocationContext.INNER,
   javax.safetycritical.annotate.AllocationContext.OUTER})
@SCJMaySelfSuspend(true)
@SCJPhase({
   javax.safetycritical.annotate.Phase.STARTUP,
```

```
    javax.safetycritical.annotate.Phase.INITIALIZATION,
    javax.safetycritical.annotate.Phase.RUN,
    javax.safetycritical.annotate.Phase.CLEANUP })
public final int readUnsignedShort( )
```

See the general contract of the readUnsignedShort method of DataInput. Bytes for this operation are read from the contained input stream.

Throws IOException

```
@SCJAllowed
@SCJMayAllocate({
    javax.safetycritical.annotate.AllocationContext.CURRENT,
    javax.safetycritical.annotate.AllocationContext.INNER,
    javax.safetycritical.annotate.AllocationContext.OUTER})
@SCJMaySelfSuspend(true)
@SCJPhase({
    javax.safetycritical.annotate.Phase.STARTUP,
    javax.safetycritical.annotate.Phase.INITIALIZATION,
    javax.safetycritical.annotate.Phase.RUN,
    javax.safetycritical.annotate.Phase.CLEANUP })
public void reset( )
```

Repositions this stream to the position at the time the mark method was last called on this input stream. This method simply performs in.reset(). Stream marks are intended to be used in situations where you need to read ahead a little to see what's in the stream. Often this is most easily done by invoking some general parser. If the stream is of the type handled by the parse, it just chugs along happily. If the stream is not of that type, the parser should toss an exception when it fails. If this happens within readlimit bytes, it allows the outer code to reset the stream and try another parser.

Throws IOException

```
@SCJAllowed
@SCJMayAllocate({
    javax.safetycritical.annotate.AllocationContext.CURRENT,
    javax.safetycritical.annotate.AllocationContext.INNER,
    javax.safetycritical.annotate.AllocationContext.OUTER})
@SCJMaySelfSuspend(true)
@SCJPhase({
    javax.safetycritical.annotate.Phase.STARTUP,
    javax.safetycritical.annotate.Phase.INITIALIZATION,
    javax.safetycritical.annotate.Phase.RUN,
    javax.safetycritical.annotate.Phase.CLEANUP })
public long skip(long n)
  throws java.io.IOException
```

Skips over and discards n bytes of data from the input stream. The skip method may, for a variety of reasons, end up skipping over some smaller number of bytes, possibly 0. The actual number of bytes skipped is returned. This method simply performs in.skip(n).

Throws IOException

```
@SCJAllowed
@SCJMayAllocate({
   javax.safetycritical.annotate.AllocationContext.CURRENT,
   javax.safetycritical.annotate.AllocationContext.INNER,
   javax.safetycritical.annotate.AllocationContext.OUTER})
@SCJMaySelfSuspend(true)
@SCJPhase({
   javax.safetycritical.annotate.Phase.STARTUP,
   javax.safetycritical.annotate.Phase.INITIALIZATION,
   javax.safetycritical.annotate.Phase.RUN,
   javax.safetycritical.annotate.Phase.CLEANUP })
public final int skipBytes(int n)
  throws java.io.IOException
```

See the general contract of the skipBytes method of DataInput. Bytes for this operation are read from the contained input stream.

Throws IOException

A.3.2 CLASS DataOutputStream

```
@SCJAllowed
public class DataOutputStream implements java.io.DataOutput extends
    java.io.OutputStream
```

A data output stream lets an application write primitive Java data types to an output stream in a portable way. An application can then use a data input stream to read the data back in.

Fields

```
@SCJAllowed
protected java.io.OutputStream out
```

Constructors

```
@SCJAllowed
@SCJMayAllocate({
    javax.safetycritical.annotate.AllocationContext.CURRENT,
    javax.safetycritical.annotate.AllocationContext.INNER,
    javax.safetycritical.annotate.AllocationContext.OUTER})
@SCJMaySelfSuspend(true)
@SCJPhase({
    javax.safetycritical.annotate.Phase.STARTUP,
    javax.safetycritical.annotate.Phase.INITIALIZATION,
    javax.safetycritical.annotate.Phase.RUN,
    javax.safetycritical.annotate.Phase.CLEANUP })
public DataOutputStream(OutputStream out)
```

Methods

```
@SCJAllowed
@SCJMayAllocate({
    javax.safetycritical.annotate.AllocationContext.CURRENT,
    javax.safetycritical.annotate.AllocationContext.INNER,
    javax.safetycritical.annotate.AllocationContext.OUTER})
@SCJMaySelfSuspend(true)
@SCJPhase({
    javax.safetycritical.annotate.Phase.STARTUP,
    javax.safetycritical.annotate.Phase.INITIALIZATION,
    javax.safetycritical.annotate.Phase.RUN,
    javax.safetycritical.annotate.Phase.CLEANUP })
public void close( )
```

Closes this output stream and releases any system resources associated with the stream.

Throws IOException

```
@SCJAllowed
@SCJMayAllocate({
    javax.safetycritical.annotate.AllocationContext.CURRENT,
    javax.safetycritical.annotate.AllocationContext.INNER,
    javax.safetycritical.annotate.AllocationContext.OUTER})
@SCJMaySelfSuspend(true)
@SCJPhase({
    javax.safetycritical.annotate.Phase.STARTUP,
    javax.safetycritical.annotate.Phase.INITIALIZATION,
    javax.safetycritical.annotate.Phase.RUN,
    javax.safetycritical.annotate.Phase.CLEANUP })
public void flush( )
```

Flushes this data output stream.

Throws IOException

```
@SCJAllowed
@SCJMayAllocate({
   javax.safetycritical.annotate.AllocationContext.CURRENT,
   javax.safetycritical.annotate.AllocationContext.INNER,
   javax.safetycritical.annotate.AllocationContext.OUTER})
@SCJMaySelfSuspend(true)
@SCJPhase({
   javax.safetycritical.annotate.Phase.STARTUP,
   javax.safetycritical.annotate.Phase.INITIALIZATION,
   javax.safetycritical.annotate.Phase.RUN,
   javax.safetycritical.annotate.Phase.CLEANUP })
public void write(int b)
  throws java.io.IOException
```

Writes the specified byte (the low eight bits of the argument b) to the underlying output stream.

Throws IOException

```
@SCJAllowed
@SCJMayAllocate({
   javax.safetycritical.annotate.AllocationContext.CURRENT,
   javax.safetycritical.annotate.AllocationContext.INNER,
   javax.safetycritical.annotate.AllocationContext.OUTER})
@SCJMaySelfSuspend(true)
@SCJPhase({
   javax.safetycritical.annotate.Phase.STARTUP,
   javax.safetycritical.annotate.Phase.INITIALIZATION,
   javax.safetycritical.annotate.Phase.RUN,
   javax.safetycritical.annotate.Phase.CLEANUP })
public void write(byte [] b, int off, int len)
  throws java.io.IOException
```

Writes len bytes from the specified byte array starting at offset off to the underlying output stream.

Throws IOException

```
@SCJAllowed
@SCJMayAllocate({
   javax.safetycritical.annotate.AllocationContext.CURRENT,
   javax.safetycritical.annotate.AllocationContext.INNER,
   javax.safetycritical.annotate.AllocationContext.OUTER})
@SCJMaySelfSuspend(true)
@SCJPhase({
```

```
    javax.safetycritical.annotate.Phase.STARTUP,
    javax.safetycritical.annotate.Phase.INITIALIZATION,
    javax.safetycritical.annotate.Phase.RUN,
    javax.safetycritical.annotate.Phase.CLEANUP })
public void writeBoolean(boolean v)
  throws java.io.IOException
```

Writes a boolean to the underlying output stream as a 1-byte value.

Throws IOException

```
@SCJAllowed
@SCJMayAllocate({
    javax.safetycritical.annotate.AllocationContext.CURRENT,
    javax.safetycritical.annotate.AllocationContext.INNER,
    javax.safetycritical.annotate.AllocationContext.OUTER})
@SCJMaySelfSuspend(true)
@SCJPhase({
    javax.safetycritical.annotate.Phase.STARTUP,
    javax.safetycritical.annotate.Phase.INITIALIZATION,
    javax.safetycritical.annotate.Phase.RUN,
    javax.safetycritical.annotate.Phase.CLEANUP })
public void writeByte(int v)
  throws java.io.IOException
```

Writes out a byte to the underlying output stream as a 1-byte value.

Throws IOException

```
@SCJAllowed
@SCJMayAllocate({
    javax.safetycritical.annotate.AllocationContext.CURRENT,
    javax.safetycritical.annotate.AllocationContext.INNER,
    javax.safetycritical.annotate.AllocationContext.OUTER})
@SCJMaySelfSuspend(true)
@SCJPhase({
    javax.safetycritical.annotate.Phase.STARTUP,
    javax.safetycritical.annotate.Phase.INITIALIZATION,
    javax.safetycritical.annotate.Phase.RUN,
    javax.safetycritical.annotate.Phase.CLEANUP })
public void writeChar(int v)
  throws java.io.IOException
```

Writes a char to the underlying output stream as a 2-byte value, high byte first.

Throws IOException

```
@SCJAllowed
@SCJMayAllocate({
    javax.safetycritical.annotate.AllocationContext.CURRENT,
    javax.safetycritical.annotate.AllocationContext.INNER,
    javax.safetycritical.annotate.AllocationContext.OUTER})
@SCJMaySelfSuspend(true)
@SCJPhase({
    javax.safetycritical.annotate.Phase.STARTUP,
    javax.safetycritical.annotate.Phase.INITIALIZATION,
    javax.safetycritical.annotate.Phase.RUN,
    javax.safetycritical.annotate.Phase.CLEANUP })
public void writeChars(String s)
  throws java.io.IOException
```

Writes a string to the underlying output stream as a sequence of characters.

Throws IOException

```
@SCJAllowed
@SCJMayAllocate({
    javax.safetycritical.annotate.AllocationContext.CURRENT,
    javax.safetycritical.annotate.AllocationContext.INNER,
    javax.safetycritical.annotate.AllocationContext.OUTER})
@SCJMaySelfSuspend(true)
@SCJPhase({
    javax.safetycritical.annotate.Phase.STARTUP,
    javax.safetycritical.annotate.Phase.INITIALIZATION,
    javax.safetycritical.annotate.Phase.RUN,
    javax.safetycritical.annotate.Phase.CLEANUP })
public void writeDouble(double v)
  throws java.io.IOException
```

Converts the double argument to a long using the doubleToLongBits method in class Double, and then writes that long value to the underlying output stream as an 8-byte quantity, high byte first.

Throws IOException

```
@SCJAllowed
@SCJMayAllocate({
    javax.safetycritical.annotate.AllocationContext.CURRENT,
    javax.safetycritical.annotate.AllocationContext.INNER,
    javax.safetycritical.annotate.AllocationContext.OUTER})
@SCJMaySelfSuspend(true)
@SCJPhase({
    javax.safetycritical.annotate.Phase.STARTUP,
    javax.safetycritical.annotate.Phase.INITIALIZATION,
    javax.safetycritical.annotate.Phase.RUN,
```

```
    javax.safetycritical.annotate.Phase.CLEANUP })
public void writeFloat(float v)
  throws java.io.IOException
```

Converts the float argument to an int using the floatToIntBits method in class Float, and then writes that int value to the underlying output stream as a 4-byte quantity, high byte first.

Throws IOException

```
@SCJAllowed
@SCJMayAllocate({
    javax.safetycritical.annotate.AllocationContext.CURRENT,
    javax.safetycritical.annotate.AllocationContext.INNER,
    javax.safetycritical.annotate.AllocationContext.OUTER})
@SCJMaySelfSuspend(true)
@SCJPhase({
    javax.safetycritical.annotate.Phase.STARTUP,
    javax.safetycritical.annotate.Phase.INITIALIZATION,
    javax.safetycritical.annotate.Phase.RUN,
    javax.safetycritical.annotate.Phase.CLEANUP })
public void writeInt(int v)
  throws java.io.IOException
```

Writes an int to the underlying output stream as four bytes, high byte first.

Throws IOException

```
@SCJAllowed
@SCJMayAllocate({
    javax.safetycritical.annotate.AllocationContext.CURRENT,
    javax.safetycritical.annotate.AllocationContext.INNER,
    javax.safetycritical.annotate.AllocationContext.OUTER})
@SCJMaySelfSuspend(true)
@SCJPhase({
    javax.safetycritical.annotate.Phase.STARTUP,
    javax.safetycritical.annotate.Phase.INITIALIZATION,
    javax.safetycritical.annotate.Phase.RUN,
    javax.safetycritical.annotate.Phase.CLEANUP })
public void writeLong(long v)
  throws java.io.IOException
```

Writes a long to the underlying output stream as eight bytes, high byte first.

Throws IOException

```
@SCJAllowed
@SCJMayAllocate({
    javax.safetycritical.annotate.AllocationContext.CURRENT,
    javax.safetycritical.annotate.AllocationContext.INNER,
    javax.safetycritical.annotate.AllocationContext.OUTER})
@SCJMaySelfSuspend(true)
@SCJPhase({
    javax.safetycritical.annotate.Phase.STARTUP,
    javax.safetycritical.annotate.Phase.INITIALIZATION,
    javax.safetycritical.annotate.Phase.RUN,
    javax.safetycritical.annotate.Phase.CLEANUP })
public void writeShort(int v)
  throws java.io.IOException
```

Writes a short to the underlying output stream as two bytes, high byte first.

Throws IOException

```
@SCJAllowed
@SCJMayAllocate({
    javax.safetycritical.annotate.AllocationContext.CURRENT,
    javax.safetycritical.annotate.AllocationContext.INNER,
    javax.safetycritical.annotate.AllocationContext.OUTER})
@SCJMaySelfSuspend(true)
@SCJPhase({
    javax.safetycritical.annotate.Phase.STARTUP,
    javax.safetycritical.annotate.Phase.INITIALIZATION,
    javax.safetycritical.annotate.Phase.RUN,
    javax.safetycritical.annotate.Phase.CLEANUP })
public void writeUTF(String str)
  throws java.io.IOException
```

Writes a string to the underlying output stream using UTF-8 encoding in a machine-independent manner.

Throws IOException

A.3.3 CLASS EOFException

```
@SCJAllowed
public class EOFException implements java.io.Serializable extends
    java.io.IOException
```

Signals that an end of file or end of stream has been reached unexpectedly during input. This exception is mainly used by data input streams to signal end of stream. Note that many other input operations return a special value on end of stream rather than throwing an exception.

Constructors

```
@SCJAllowed
@SCJPhase({
   javax.safetycritical.annotate.Phase.INITIALIZATION,
   javax.safetycritical.annotate.Phase.RUN,
   javax.safetycritical.annotate.Phase.CLEANUP})
@SCJMaySelfSuspend(false)
@SCJMayAllocate({javax.safetycritical.annotate.AllocationContext.CURRENT})
public EOFException( )
```

Shall not copy ”this” to any instance or static field.

Invokes System.captureStackBacktrace(this) to save the backtrace associated with the current thread.

```
@SCJAllowed
@SCJPhase({
   javax.safetycritical.annotate.Phase.INITIALIZATION,
   javax.safetycritical.annotate.Phase.RUN,
   javax.safetycritical.annotate.Phase.CLEANUP})
@SCJMaySelfSuspend(false)
@SCJMayAllocate({javax.safetycritical.annotate.AllocationContext.CURRENT})
public EOFException(String msg)
```

Shall not copy ”this” to any instance or static field. The scope containing the msg argument must enclose the scope containing ”this”. Otherwise, an IllegalAssignmentError will be thrown.

Invokes System.captureStackBacktrace(this) to save the backtrace associated with the current thread.

A.3.4 CLASS FilterOutputStream

```
@SCJAllowed
public class FilterOutputStream extends java.io.OutputStream
```

Unless specified to the contrary, see JDK for Java 8 documentation.

Constructors

```
@SCJAllowed
@SCJPhase({
   javax.safetycritical.annotate.Phase.INITIALIZATION,
   javax.safetycritical.annotate.Phase.RUN,
```

```
    javax.safetycritical.annotate.Phase.CLEANUP})
@SCJMaySelfSuspend(false)
@SCJMayAllocate({javax.safetycritical.annotate.AllocationContext.CURRENT})
public FilterOutputStream(OutputStream out)
```

Methods

```
@SCJAllowed
@SCJMayAllocate({
    javax.safetycritical.annotate.AllocationContext.CURRENT,
    javax.safetycritical.annotate.AllocationContext.INNER,
    javax.safetycritical.annotate.AllocationContext.OUTER})
@SCJMaySelfSuspend(true)
@SCJPhase({
    javax.safetycritical.annotate.Phase.STARTUP,
    javax.safetycritical.annotate.Phase.INITIALIZATION,
    javax.safetycritical.annotate.Phase.RUN,
    javax.safetycritical.annotate.Phase.CLEANUP })
public void close( )
```

```
@SCJAllowed
@SCJMayAllocate({
    javax.safetycritical.annotate.AllocationContext.CURRENT,
    javax.safetycritical.annotate.AllocationContext.INNER,
    javax.safetycritical.annotate.AllocationContext.OUTER})
@SCJMaySelfSuspend(true)
@SCJPhase({
    javax.safetycritical.annotate.Phase.STARTUP,
    javax.safetycritical.annotate.Phase.INITIALIZATION,
    javax.safetycritical.annotate.Phase.RUN,
    javax.safetycritical.annotate.Phase.CLEANUP })
public void flush( )
```

```
@SCJAllowed
@SCJMayAllocate({
    javax.safetycritical.annotate.AllocationContext.CURRENT,
    javax.safetycritical.annotate.AllocationContext.INNER,
    javax.safetycritical.annotate.AllocationContext.OUTER})
@SCJMaySelfSuspend(true)
@SCJPhase({
    javax.safetycritical.annotate.Phase.STARTUP,
    javax.safetycritical.annotate.Phase.INITIALIZATION,
    javax.safetycritical.annotate.Phase.RUN,
    javax.safetycritical.annotate.Phase.CLEANUP })
public void write(byte [] b)
  throws java.io.IOException
```

```
@SCJAllowed
@SCJMayAllocate({
   javax.safetycritical.annotate.AllocationContext.CURRENT,
   javax.safetycritical.annotate.AllocationContext.INNER,
   javax.safetycritical.annotate.AllocationContext.OUTER})
@SCJMaySelfSuspend(true)
@SCJPhase({
   javax.safetycritical.annotate.Phase.STARTUP,
   javax.safetycritical.annotate.Phase.INITIALIZATION,
   javax.safetycritical.annotate.Phase.RUN,
   javax.safetycritical.annotate.Phase.CLEANUP })
public void write(byte [] b, int off, int len)
  throws java.io.IOException
```

```
@SCJAllowed
@SCJMayAllocate({
   javax.safetycritical.annotate.AllocationContext.CURRENT,
   javax.safetycritical.annotate.AllocationContext.INNER,
   javax.safetycritical.annotate.AllocationContext.OUTER})
@SCJMaySelfSuspend(true)
@SCJPhase({
   javax.safetycritical.annotate.Phase.STARTUP,
   javax.safetycritical.annotate.Phase.INITIALIZATION,
   javax.safetycritical.annotate.Phase.RUN,
   javax.safetycritical.annotate.Phase.CLEANUP })
public void write(int b)
  throws java.io.IOException
```

A.3.5 CLASS IOException

```
@SCJAllowed
public class IOException implements java.io.Serializable extends
    java.lang.Exception
```

Signals that an I/O exception of some sort has occurred. This class is the general class of exceptions produced by failed or interrupted I/O operations.

Constructors

```
@SCJAllowed
@SCJPhase({
   javax.safetycritical.annotate.Phase.INITIALIZATION,
   javax.safetycritical.annotate.Phase.RUN,
   javax.safetycritical.annotate.Phase.CLEANUP})
@SCJMaySelfSuspend(false)
@SCJMayAllocate({javax.safetycritical.annotate.AllocationContext.CURRENT})
public IOException( )
```

Shall not copy ”this” to any instance or static field.

Invokes System.captureStackBacktrace(this) to save the backtrace associated with the current thread.

```
@SCJAllowed
@SCJPhase({
   javax.safetycritical.annotate.Phase.INITIALIZATION,
   javax.safetycritical.annotate.Phase.RUN,
   javax.safetycritical.annotate.Phase.CLEANUP})
@SCJMaySelfSuspend(false)
@SCJMayAllocate({javax.safetycritical.annotate.AllocationContext.CURRENT})
public IOException(String msg)
```

Shall not copy ”this” to any instance or static field. The scope containing the msg argument must enclose the scope containing ”this”. Otherwise, an IllegalAssignmentError will be thrown.

Invokes System.captureStackBacktrace(this) to save the backtrace associated with the current thread.

A.3.6 CLASS InputStream

```
@SCJAllowed
public abstract class InputStream implements java.io.Closeable extends
    java.lang.Object
```

Unless specified to the contrary, see JDK for Java 8 documentation.

Constructors

```
@SCJAllowed
@SCJPhase({
   javax.safetycritical.annotate.Phase.INITIALIZATION,
   javax.safetycritical.annotate.Phase.RUN,
   javax.safetycritical.annotate.Phase.CLEANUP})
@SCJMaySelfSuspend(false)
@SCJMayAllocate({javax.safetycritical.annotate.AllocationContext.CURRENT})
public InputStream( )
```

Methods

```
@SCJAllowed
@SCJMayAllocate({
   javax.safetycritical.annotate.AllocationContext.CURRENT,
```

```
    javax.safetycritical.annotate.AllocationContext.INNER,
    javax.safetycritical.annotate.AllocationContext.OUTER})
@SCJMaySelfSuspend(true)
@SCJPhase({
    javax.safetycritical.annotate.Phase.STARTUP,
    javax.safetycritical.annotate.Phase.INITIALIZATION,
    javax.safetycritical.annotate.Phase.RUN,
    javax.safetycritical.annotate.Phase.CLEANUP })
public int available( )
```

Throws IOException

```
@SCJAllowed
@SCJMayAllocate({
    javax.safetycritical.annotate.AllocationContext.CURRENT,
    javax.safetycritical.annotate.AllocationContext.INNER,
    javax.safetycritical.annotate.AllocationContext.OUTER})
@SCJMaySelfSuspend(true)
@SCJPhase({
    javax.safetycritical.annotate.Phase.STARTUP,
    javax.safetycritical.annotate.Phase.INITIALIZATION,
    javax.safetycritical.annotate.Phase.RUN,
    javax.safetycritical.annotate.Phase.CLEANUP })
public void close( )
```

Throws IOException

```
@SCJAllowed
@SCJMayAllocate({
    javax.safetycritical.annotate.AllocationContext.CURRENT,
    javax.safetycritical.annotate.AllocationContext.INNER,
    javax.safetycritical.annotate.AllocationContext.OUTER})
@SCJMaySelfSuspend(true)
@SCJPhase({
    javax.safetycritical.annotate.Phase.STARTUP,
    javax.safetycritical.annotate.Phase.INITIALIZATION,
    javax.safetycritical.annotate.Phase.RUN,
    javax.safetycritical.annotate.Phase.CLEANUP })
public void mark(int readlimit)
```

```
@SCJAllowed
@SCJMayAllocate({
    javax.safetycritical.annotate.AllocationContext.CURRENT,
    javax.safetycritical.annotate.AllocationContext.INNER,
    javax.safetycritical.annotate.AllocationContext.OUTER})
@SCJMaySelfSuspend(true)
@SCJPhase({
    javax.safetycritical.annotate.Phase.STARTUP,
    javax.safetycritical.annotate.Phase.INITIALIZATION,
```

```
    javax.safetycritical.annotate.Phase.RUN,
    javax.safetycritical.annotate.Phase.CLEANUP })
public boolean markSupported( )


@SCJAllowed
@SCJMayAllocate({
    javax.safetycritical.annotate.AllocationContext.CURRENT,
    javax.safetycritical.annotate.AllocationContext.INNER,
    javax.safetycritical.annotate.AllocationContext.OUTER})
@SCJMaySelfSuspend(true)
@SCJPhase({
    javax.safetycritical.annotate.Phase.STARTUP,
    javax.safetycritical.annotate.Phase.INITIALIZATION,
    javax.safetycritical.annotate.Phase.RUN,
    javax.safetycritical.annotate.Phase.CLEANUP })
public int read(byte [] b)
  throws java.io.IOException
```

Throws IOException

```
@SCJAllowed
@SCJMayAllocate({
    javax.safetycritical.annotate.AllocationContext.CURRENT,
    javax.safetycritical.annotate.AllocationContext.INNER,
    javax.safetycritical.annotate.AllocationContext.OUTER})
@SCJMaySelfSuspend(true)
@SCJPhase({
    javax.safetycritical.annotate.Phase.STARTUP,
    javax.safetycritical.annotate.Phase.INITIALIZATION,
    javax.safetycritical.annotate.Phase.RUN,
    javax.safetycritical.annotate.Phase.CLEANUP })
public int read(byte [] b, int off, int len)
  throws java.io.IOException
```

Throws IOException

```
@SCJAllowed
@SCJMayAllocate({
    javax.safetycritical.annotate.AllocationContext.CURRENT,
    javax.safetycritical.annotate.AllocationContext.INNER,
    javax.safetycritical.annotate.AllocationContext.OUTER})
@SCJMaySelfSuspend(true)
@SCJPhase({
    javax.safetycritical.annotate.Phase.STARTUP,
    javax.safetycritical.annotate.Phase.INITIALIZATION,
    javax.safetycritical.annotate.Phase.RUN,
    javax.safetycritical.annotate.Phase.CLEANUP })
public abstract int read( )
```

Throws IOException

```
@SCJAllowed
@SCJMayAllocate({
    javax.safetycritical.annotate.AllocationContext.CURRENT,
    javax.safetycritical.annotate.AllocationContext.INNER,
    javax.safetycritical.annotate.AllocationContext.OUTER})
@SCJMaySelfSuspend(true)
@SCJPhase({
    javax.safetycritical.annotate.Phase.STARTUP,
    javax.safetycritical.annotate.Phase.INITIALIZATION,
    javax.safetycritical.annotate.Phase.RUN,
    javax.safetycritical.annotate.Phase.CLEANUP })
public void reset( )
```

Throws IOException

```
@SCJAllowed
@SCJMayAllocate({
    javax.safetycritical.annotate.AllocationContext.CURRENT,
    javax.safetycritical.annotate.AllocationContext.INNER,
    javax.safetycritical.annotate.AllocationContext.OUTER})
@SCJMaySelfSuspend(true)
@SCJPhase({
    javax.safetycritical.annotate.Phase.STARTUP,
    javax.safetycritical.annotate.Phase.INITIALIZATION,
    javax.safetycritical.annotate.Phase.RUN,
    javax.safetycritical.annotate.Phase.CLEANUP })
public long skip(long n)
  throws java.io.IOException
```

Throws IOException

A.3.7 CLASS OutputStream

```
@SCJAllowed
public abstract class OutputStream implements java.io.Closeable,
      java.io.Flushable extends java.lang.Object
```

Unless specified to the contrary, see JDK for Java 8 documentation.

Constructors

```
@SCJAllowed
@SCJPhase({
    javax.safetycritical.annotate.Phase.INITIALIZATION,
    javax.safetycritical.annotate.Phase.RUN,
```

```
    javax.safetycritical.annotate.Phase.CLEANUP})
@SCJMaySelfSuspend(false)
@SCJMayAllocate({javax.safetycritical.annotate.AllocationContext.CURRENT})
public OutputStream( )
```

Methods

```
@SCJAllowed
@SCJMayAllocate({
    javax.safetycritical.annotate.AllocationContext.CURRENT,
    javax.safetycritical.annotate.AllocationContext.INNER,
    javax.safetycritical.annotate.AllocationContext.OUTER})
@SCJMaySelfSuspend(true)
@SCJPhase({
    javax.safetycritical.annotate.Phase.STARTUP,
    javax.safetycritical.annotate.Phase.INITIALIZATION,
    javax.safetycritical.annotate.Phase.RUN,
    javax.safetycritical.annotate.Phase.CLEANUP })
public void close( )
```

Throws IOException

```
@SCJAllowed
@SCJMayAllocate({
    javax.safetycritical.annotate.AllocationContext.CURRENT,
    javax.safetycritical.annotate.AllocationContext.INNER,
    javax.safetycritical.annotate.AllocationContext.OUTER})
@SCJMaySelfSuspend(true)
@SCJPhase({
    javax.safetycritical.annotate.Phase.STARTUP,
    javax.safetycritical.annotate.Phase.INITIALIZATION,
    javax.safetycritical.annotate.Phase.RUN,
    javax.safetycritical.annotate.Phase.CLEANUP })
public void flush( )
```

Throws IOException

```
@SCJAllowed
@SCJMayAllocate({
    javax.safetycritical.annotate.AllocationContext.CURRENT,
    javax.safetycritical.annotate.AllocationContext.INNER,
    javax.safetycritical.annotate.AllocationContext.OUTER})
@SCJMaySelfSuspend(true)
@SCJPhase({
    javax.safetycritical.annotate.Phase.STARTUP,
    javax.safetycritical.annotate.Phase.INITIALIZATION,
    javax.safetycritical.annotate.Phase.RUN,
    javax.safetycritical.annotate.Phase.CLEANUP })
public void write(byte [] b)
```

throws java.io.IOException

Throws IOException

```
@SCJAllowed
@SCJMayAllocate({
    javax.safetycritical.annotate.AllocationContext.CURRENT,
    javax.safetycritical.annotate.AllocationContext.INNER,
    javax.safetycritical.annotate.AllocationContext.OUTER})
@SCJMaySelfSuspend(true)
@SCJPhase({
    javax.safetycritical.annotate.Phase.STARTUP,
    javax.safetycritical.annotate.Phase.INITIALIZATION,
    javax.safetycritical.annotate.Phase.RUN,
    javax.safetycritical.annotate.Phase.CLEANUP })
public void write(byte [] b, int off, int len)
  throws java.io.IOException
```

Throws IOException

```
@SCJAllowed
@SCJMayAllocate({
    javax.safetycritical.annotate.AllocationContext.CURRENT,
    javax.safetycritical.annotate.AllocationContext.INNER,
    javax.safetycritical.annotate.AllocationContext.OUTER})
@SCJMaySelfSuspend(true)
@SCJPhase({
    javax.safetycritical.annotate.Phase.STARTUP,
    javax.safetycritical.annotate.Phase.INITIALIZATION,
    javax.safetycritical.annotate.Phase.RUN,
    javax.safetycritical.annotate.Phase.CLEANUP })
public abstract void write(int b)
  throws java.io.IOException
```

Throws IOException

A.3.8 CLASS PrintStream

```
@SCJAllowed
public class PrintStream extends java.io.OutputStream
```

A PrintStream adds functionality to an output stream, namely the ability to print representations of various data values conveniently. A PrintStream never throws an IOException; instead, exceptional situations merely set an internal flag that can be tested via the checkError method. Optionally, a PrintStream can be created to flush automatically; this means that the flush method is automatically invoked after a byte array is written, one of the println methods is

invoked, or a newline character or byte ('\n') is written.

All characters printed by a PrintStream are converted into bytes using the platform's default character encoding.

Constructors

```
@SCJAllowed
@SCJMayAllocate({})
@SCJMaySelfSuspend(true)
@SCJPhase({
   javax.safetycritical.annotate.Phase.STARTUP,
   javax.safetycritical.annotate.Phase.INITIALIZATION,
   javax.safetycritical.annotate.Phase.RUN,
   javax.safetycritical.annotate.Phase.CLEANUP })
public PrintStream(OutputStream out)
```

Create a new print stream. This stream will not flush automatically.

out — The output stream to which values and objects will be printed.

Methods

```
@SCJAllowed
@SCJMaySelfSuspend(true)
@SCJMayAllocate({
   javax.safetycritical.annotate.AllocationContext.CURRENT,
   javax.safetycritical.annotate.AllocationContext.INNER,
   javax.safetycritical.annotate.AllocationContext.OUTER})
@SCJPhase({
   javax.safetycritical.annotate.Phase.STARTUP,
   javax.safetycritical.annotate.Phase.INITIALIZATION,
   javax.safetycritical.annotate.Phase.RUN,
   javax.safetycritical.annotate.Phase.CLEANUP })
public boolean checkError( )
```

Flush the stream and check its error state. The internal error state is set to true when the underlying output stream throws an IOException, and when the setError method is invoked.

returns true if and only if this stream has encountered an IOException, or the setError method has been invoked.

```
@SCJAllowed
@SCJMaySelfSuspend(true)
@SCJMayAllocate({
```

```
    javax.safetycritical.annotate.AllocationContext.CURRENT,
    javax.safetycritical.annotate.AllocationContext.INNER,
    javax.safetycritical.annotate.AllocationContext.OUTER})
@SCJPhase({
    javax.safetycritical.annotate.Phase.STARTUP,
    javax.safetycritical.annotate.Phase.INITIALIZATION,
    javax.safetycritical.annotate.Phase.RUN,
    javax.safetycritical.annotate.Phase.CLEANUP })
public void close( )
```

Close the stream. This is done by flushing the stream and then closing the underlying output stream.

See Also: java.io.OutputStream.close()

```
@SCJAllowed
@SCJMaySelfSuspend(true)
@SCJMayAllocate({
    javax.safetycritical.annotate.AllocationContext.CURRENT,
    javax.safetycritical.annotate.AllocationContext.INNER,
    javax.safetycritical.annotate.AllocationContext.OUTER})
@SCJPhase({
    javax.safetycritical.annotate.Phase.STARTUP,
    javax.safetycritical.annotate.Phase.INITIALIZATION,
    javax.safetycritical.annotate.Phase.RUN,
    javax.safetycritical.annotate.Phase.CLEANUP })
public void flush( )
```

Flush the stream. This is done by writing any buffered output bytes to the underlying output stream and then flushing that stream.

See Also: java.io.OutputStream.flush()

```
@SCJAllowed
@SCJMaySelfSuspend(false)
@SCJMayAllocate({
    javax.safetycritical.annotate.AllocationContext.CURRENT,
    javax.safetycritical.annotate.AllocationContext.INNER,
    javax.safetycritical.annotate.AllocationContext.OUTER})
@SCJPhase({
    javax.safetycritical.annotate.Phase.STARTUP,
    javax.safetycritical.annotate.Phase.INITIALIZATION,
    javax.safetycritical.annotate.Phase.RUN,
    javax.safetycritical.annotate.Phase.CLEANUP })
public void print(int i)
```

Print an integer. The string produced by java.lang.String.valueOf(i) is translated into bytes according to the platform's default character encoding, and these bytes are written in exactly the manner of the write() method.

i — The int to be printed.

See Also: java.lang.Integer.toString(int)

```
@SCJAllowed
@SCJMaySelfSuspend(false)
@SCJMayAllocate({
   javax.safetycritical.annotate.AllocationContext.CURRENT,
   javax.safetycritical.annotate.AllocationContext.INNER,
   javax.safetycritical.annotate.AllocationContext.OUTER})
@SCJPhase({
   javax.safetycritical.annotate.Phase.STARTUP,
   javax.safetycritical.annotate.Phase.INITIALIZATION,
   javax.safetycritical.annotate.Phase.RUN,
   javax.safetycritical.annotate.Phase.CLEANUP })
public void print(char [] s)
```

Print an array of characters. The characters are converted into bytes according to the platform's default character encoding, and these bytes are written in exactly the manner of the write() method.

s — The array of chars to be printed.

Throws NullPointerException If s is null

```
@SCJAllowed
@SCJMaySelfSuspend(false)
@SCJMayAllocate({
   javax.safetycritical.annotate.AllocationContext.CURRENT,
   javax.safetycritical.annotate.AllocationContext.INNER,
   javax.safetycritical.annotate.AllocationContext.OUTER})
@SCJPhase({
   javax.safetycritical.annotate.Phase.STARTUP,
   javax.safetycritical.annotate.Phase.INITIALIZATION,
   javax.safetycritical.annotate.Phase.RUN,
   javax.safetycritical.annotate.Phase.CLEANUP })
public void print(Object obj)
```

Print an object. The string produced by the java.lang.String.valueOf(obj) method is translated into bytes according to the platform's default character encoding, and these bytes are written in exactly the manner of the write() method.

obj — The Object to be printed.

See Also: java.lang.Object.toString()

```
@SCJAllowed
@SCJMaySelfSuspend(false)
```

```
@SCJMayAllocate({
   javax.safetycritical.annotate.AllocationContext.CURRENT,
   javax.safetycritical.annotate.AllocationContext.INNER,
   javax.safetycritical.annotate.AllocationContext.OUTER})
@SCJPhase({
   javax.safetycritical.annotate.Phase.STARTUP,
   javax.safetycritical.annotate.Phase.INITIALIZATION,
   javax.safetycritical.annotate.Phase.RUN,
   javax.safetycritical.annotate.Phase.CLEANUP })
public void print(String s)
```

Print a string. If the argument is null then the string "null" is printed. Otherwise, the string's characters are converted into bytes according to the platform's default character encoding, and these bytes are written in exactly the manner of the write() method.

s — The String to be printed.

```
@SCJAllowed
@SCJMaySelfSuspend(false)
@SCJMayAllocate({
   javax.safetycritical.annotate.AllocationContext.CURRENT,
   javax.safetycritical.annotate.AllocationContext.INNER,
   javax.safetycritical.annotate.AllocationContext.OUTER})
@SCJPhase({
   javax.safetycritical.annotate.Phase.STARTUP,
   javax.safetycritical.annotate.Phase.INITIALIZATION,
   javax.safetycritical.annotate.Phase.RUN,
   javax.safetycritical.annotate.Phase.CLEANUP })
public void print(long l)
```

Print a long integer. The string produced by java.lang.String.valueOf(l) is translated into bytes according to the platform's default character encoding, and these bytes are written in exactly the manner of the write() method.

l — The long to be printed.

See Also: java.lang.Long.toString(long)

```
@SCJAllowed
@SCJMaySelfSuspend(false)
@SCJMayAllocate({
   javax.safetycritical.annotate.AllocationContext.CURRENT,
   javax.safetycritical.annotate.AllocationContext.INNER,
   javax.safetycritical.annotate.AllocationContext.OUTER})
@SCJPhase({
   javax.safetycritical.annotate.Phase.STARTUP,
```

```
    javax.safetycritical.annotate.Phase.INITIALIZATION,
    javax.safetycritical.annotate.Phase.RUN,
    javax.safetycritical.annotate.Phase.CLEANUP })
public void print(char c)
```

Print a character. The character is translated into one or more bytes according to the platform's default character encoding, and these bytes are written in exactly the manner of the write() method.

c — The char to be printed.

```
@SCJAllowed
@SCJMaySelfSuspend(false)
@SCJMayAllocate({
    javax.safetycritical.annotate.AllocationContext.CURRENT,
    javax.safetycritical.annotate.AllocationContext.INNER,
    javax.safetycritical.annotate.AllocationContext.OUTER})
@SCJPhase({
    javax.safetycritical.annotate.Phase.STARTUP,
    javax.safetycritical.annotate.Phase.INITIALIZATION,
    javax.safetycritical.annotate.Phase.RUN,
    javax.safetycritical.annotate.Phase.CLEANUP })
public void print(boolean b)
```

Print a boolean value. The string produced by java.lang.String.valueOf(b) is translated into bytes according to the platform's default character encoding, and these bytes are written in exactly the manner of the write() method.

b — The boolean to be printed.

```
@SCJAllowed
@SCJMaySelfSuspend(true)
@SCJMayAllocate({
    javax.safetycritical.annotate.AllocationContext.CURRENT,
    javax.safetycritical.annotate.AllocationContext.INNER,
    javax.safetycritical.annotate.AllocationContext.OUTER})
@SCJPhase({
    javax.safetycritical.annotate.Phase.STARTUP,
    javax.safetycritical.annotate.Phase.INITIALIZATION,
    javax.safetycritical.annotate.Phase.RUN,
    javax.safetycritical.annotate.Phase.CLEANUP })
public void println(boolean x)
```

Print a boolean and then terminate the line. This method behaves as though it invokes print(x) and then println().

x — The boolean to be printed.

```
@SCJAllowed
@SCJMaySelfSuspend(true)
@SCJMayAllocate({
   javax.safetycritical.annotate.AllocationContext.CURRENT,
   javax.safetycritical.annotate.AllocationContext.INNER,
   javax.safetycritical.annotate.AllocationContext.OUTER})
@SCJPhase({
   javax.safetycritical.annotate.Phase.STARTUP,
   javax.safetycritical.annotate.Phase.INITIALIZATION,
   javax.safetycritical.annotate.Phase.RUN,
   javax.safetycritical.annotate.Phase.CLEANUP })
public void println(char x)
```

Print a character and then terminate the line. This method behaves as though it invokes print(x) and then println().

x — The char to be printed.

```
@SCJAllowed
@SCJMaySelfSuspend(true)
@SCJMayAllocate({
   javax.safetycritical.annotate.AllocationContext.CURRENT,
   javax.safetycritical.annotate.AllocationContext.INNER,
   javax.safetycritical.annotate.AllocationContext.OUTER})
@SCJPhase({
   javax.safetycritical.annotate.Phase.STARTUP,
   javax.safetycritical.annotate.Phase.INITIALIZATION,
   javax.safetycritical.annotate.Phase.RUN,
   javax.safetycritical.annotate.Phase.CLEANUP })
public void println(int x)
```

Print an integer and then terminate the line. This method behaves as though it invokes print(x) and then println().

x — The int to be printed.

```
@SCJAllowed
@SCJMaySelfSuspend(true)
@SCJMayAllocate({
   javax.safetycritical.annotate.AllocationContext.CURRENT,
   javax.safetycritical.annotate.AllocationContext.INNER,
   javax.safetycritical.annotate.AllocationContext.OUTER})
@SCJPhase({
   javax.safetycritical.annotate.Phase.STARTUP,
   javax.safetycritical.annotate.Phase.INITIALIZATION,
   javax.safetycritical.annotate.Phase.RUN,
   javax.safetycritical.annotate.Phase.CLEANUP })
public void println(char [] x)
```

Print an array of characters and then terminate the line. This method behaves as though it invokes print(x) and then println().

x — an array of chars to print.

```
@SCJAllowed
@SCJMaySelfSuspend(true)
@SCJMayAllocate({
   javax.safetycritical.annotate.AllocationContext.CURRENT,
   javax.safetycritical.annotate.AllocationContext.INNER,
   javax.safetycritical.annotate.AllocationContext.OUTER})
@SCJPhase({
   javax.safetycritical.annotate.Phase.STARTUP,
   javax.safetycritical.annotate.Phase.INITIALIZATION,
   javax.safetycritical.annotate.Phase.RUN,
   javax.safetycritical.annotate.Phase.CLEANUP })
public void println(String x)
```

Print a String and then terminate the line. This method behaves as though it invokes print(x) and then println().

x — The String to be printed.

```
@SCJAllowed
@SCJMaySelfSuspend(true)
@SCJMayAllocate({
   javax.safetycritical.annotate.AllocationContext.CURRENT,
   javax.safetycritical.annotate.AllocationContext.INNER,
   javax.safetycritical.annotate.AllocationContext.OUTER})
@SCJPhase({
   javax.safetycritical.annotate.Phase.STARTUP,
   javax.safetycritical.annotate.Phase.INITIALIZATION,
   javax.safetycritical.annotate.Phase.RUN,
   javax.safetycritical.annotate.Phase.CLEANUP })
public void println(Object x)
```

Print an Object and then terminate the line. This method behaves as though it invokes print(x) and then println().

x — The Object to be printed.

```
@SCJAllowed
@SCJMaySelfSuspend(true)
@SCJMayAllocate({
   javax.safetycritical.annotate.AllocationContext.CURRENT,
   javax.safetycritical.annotate.AllocationContext.INNER,
   javax.safetycritical.annotate.AllocationContext.OUTER})
```

```
@SCJPhase({
    javax.safetycritical.annotate.Phase.STARTUP,
    javax.safetycritical.annotate.Phase.INITIALIZATION,
    javax.safetycritical.annotate.Phase.RUN,
    javax.safetycritical.annotate.Phase.CLEANUP })
public void println(long x)
```

Print a long and then terminate the line. This method behaves as though it invokes print(x) and then println().

x — a The long to be printed.

```
@SCJAllowed
@SCJMaySelfSuspend(true)
@SCJMayAllocate({
    javax.safetycritical.annotate.AllocationContext.CURRENT,
    javax.safetycritical.annotate.AllocationContext.INNER,
    javax.safetycritical.annotate.AllocationContext.OUTER})
@SCJPhase({
    javax.safetycritical.annotate.Phase.STARTUP,
    javax.safetycritical.annotate.Phase.INITIALIZATION,
    javax.safetycritical.annotate.Phase.RUN,
    javax.safetycritical.annotate.Phase.CLEANUP })
public void println( )
```

Terminate the current line by writing the line separator string. The line separator string is defined by the system property line.separator, and is not necessarily a single newline character ('\n').

```
@SCJAllowed
@SCJMayAllocate({})
@SCJMaySelfSuspend(false)
@SCJPhase({
    javax.safetycritical.annotate.Phase.STARTUP,
    javax.safetycritical.annotate.Phase.INITIALIZATION,
    javax.safetycritical.annotate.Phase.RUN,
    javax.safetycritical.annotate.Phase.CLEANUP })
protected void setError( )
```

Set the error state of the stream to true.

Since
JDK1.1

```
@SCJAllowed
@SCJMaySelfSuspend(true)
@SCJMayAllocate({
   javax.safetycritical.annotate.AllocationContext.CURRENT,
   javax.safetycritical.annotate.AllocationContext.INNER,
   javax.safetycritical.annotate.AllocationContext.OUTER})
@SCJPhase({
   javax.safetycritical.annotate.Phase.STARTUP,
   javax.safetycritical.annotate.Phase.INITIALIZATION,
   javax.safetycritical.annotate.Phase.RUN,
   javax.safetycritical.annotate.Phase.CLEANUP })
public void write(byte [] buf, int off, int len)
```

Write len bytes from the specified byte array starting at offset off to this stream. If automatic flushing is enabled then the flush method will be invoked.

Note that the bytes will be written as given; to write characters that will be translated according to the platform's default character encoding, use the print() or println() methods.

buf — A byte array.

off — Offset from which to start taking bytes.

len — Number of bytes to write.

```
@SCJAllowed
@SCJMaySelfSuspend(true)
@SCJMayAllocate({
   javax.safetycritical.annotate.AllocationContext.CURRENT,
   javax.safetycritical.annotate.AllocationContext.INNER,
   javax.safetycritical.annotate.AllocationContext.OUTER})
@SCJPhase({
   javax.safetycritical.annotate.Phase.STARTUP,
   javax.safetycritical.annotate.Phase.INITIALIZATION,
   javax.safetycritical.annotate.Phase.RUN,
   javax.safetycritical.annotate.Phase.CLEANUP })
public void write(int b)
```

Write the specified byte to this stream. If the byte is a newline and automatic flushing is enabled then the flush method will be invoked.

Note that the byte is written as given; to write a character that will be translated according to the platform's default character encoding, use the print() or println() methods.

b — The byte to be written.

See Also: java.io.PrintStream.print(char), java.io.PrintStream.println(char)

A.3.9 CLASS **UTFDataFormatException**

```
@SCJAllowed
public class UTFDataFormatException implements java.io.Serializable extends
    java.io.IOException
```

Signals that a malformed string in modified UTF-8 format has been read in a data input stream or by any class that implements the data input interface. See the DataInput class description for the format in which modified UTF-8 strings are read and written.

Constructors

```
@SCJAllowed
@SCJPhase({
   javax.safetycritical.annotate.Phase.INITIALIZATION,
   javax.safetycritical.annotate.Phase.RUN,
   javax.safetycritical.annotate.Phase.CLEANUP})
@SCJMaySelfSuspend(false)
@SCJMayAllocate({javax.safetycritical.annotate.AllocationContext.CURRENT})
public UTFDataFormatException( )
```

Shall not copy ”this” to any instance or static field.

Invokes System.captureStackBacktrace(this) to save the backtrace associated with the current thread.

```
@SCJAllowed
@SCJPhase({
   javax.safetycritical.annotate.Phase.INITIALIZATION,
   javax.safetycritical.annotate.Phase.RUN,
   javax.safetycritical.annotate.Phase.CLEANUP})
@SCJMaySelfSuspend(false)
@SCJMayAllocate({javax.safetycritical.annotate.AllocationContext.CURRENT})
public UTFDataFormatException(String msg)
```

Shall not copy ”this” to any instance or static field. The scope containing the msg argument must enclose the scope containing ”this”. Otherwise, an IllegalAssignmentError will be thrown.

Invokes System.captureStackBacktrace(this) to save the backtrace associated with the current thread.

Appendix B

Javadoc Description of Package java.lang

Package Contents *Page*

B.1 Classes

B.1.1 CLASS Deprecated

```
@SCJAllowed @Documented @Retention(java.lang.annotation.RetentionPolicy.RUNTIME)
public @interface Deprecated
```

A program element annotated @Deprecated is one that programmers are discouraged from using, typically because it is dangerous, or because a better alternative exists. Compilers warn when a deprecated program element is used or overridden in non-deprecated code.

B.1.2 CLASS Override

```
@Documented
@Retention(java.lang.annotation.RetentionPolicy.SOURCE)
@SCJAllowed
@Target({java.lang.annotation.ElementType.METHOD})
public @interface Override
```

Indicates that a method declaration is intended to override a method declaration in a supertype. If a method is annotated with this annotation type compilers are required to generate an error message unless the method does override or implement a method declared in a supertype, or the method has a signature that is override-equivalent to that of any public method declared in Object.

B.1.3 CLASS SuppressWarnings

```
@Retention(java.lang.annotation.RetentionPolicy.SOURCE)
@SCJAllowed
@Target({
   java.lang.annotation.ElementType.TYPE, java.lang.annotation.ElementType.FIELD,
   java.lang.annotation.ElementType.METHOD,
   java.lang.annotation.ElementType.PARAMETER,
   java.lang.annotation.ElementType.CONSTRUCTOR,
   java.lang.annotation.ElementType.LOCAL_VARIABLE})
public @interface SuppressWarnings
```

Indicates that the named compiler warnings should be suppressed in the annotated element (and in all program elements contained in the annotated element). Note that the set of warnings suppressed in a given element is a superset of the warnings suppressed in all containing elements. For example, if you annotate a class to suppress one warning and annotate a method to suppress another, both warnings will be suppressed in the method.

B.2 Interfaces

B.2.1 INTERFACE Appendable

```
@SCJAllowed
public interface Appendable
```

An object to which char sequences and values can be appended.

Methods

```
@SCJMayAllocate({javax.safetycritical.annotate.AllocationContext.THIS})
@SCJAllowed
@SCJMaySelfSuspend(true)
@SCJPhase({
    javax.safetycritical.annotate.Phase.STARTUP,
    javax.safetycritical.annotate.Phase.INITIALIZATION,
    javax.safetycritical.annotate.Phase.RUN,
    javax.safetycritical.annotate.Phase.CLEANUP })
public java.lang.Appendable append(CharSequence csq)
```

```
@SCJMayAllocate({javax.safetycritical.annotate.AllocationContext.THIS})
@SCJAllowed
@SCJMaySelfSuspend(true)
@SCJPhase({
    javax.safetycritical.annotate.Phase.STARTUP,
    javax.safetycritical.annotate.Phase.INITIALIZATION,
    javax.safetycritical.annotate.Phase.RUN,
    javax.safetycritical.annotate.Phase.CLEANUP })
public java.lang.Appendable append(CharSequence csq, int start, int end)
```

```
@SCJMayAllocate({javax.safetycritical.annotate.AllocationContext.THIS})
@SCJAllowed
@SCJMaySelfSuspend(true)
@SCJPhase({
    javax.safetycritical.annotate.Phase.STARTUP,
    javax.safetycritical.annotate.Phase.INITIALIZATION,
    javax.safetycritical.annotate.Phase.RUN,
    javax.safetycritical.annotate.Phase.CLEANUP })
public java.lang.Appendable append(char c)
```

B.2.2 INTERFACE CharSequence

```
@SCJAllowed
public interface CharSequence
```

A CharSequence is a readable sequence of char values. This interface provides uniform, read-only access to many different kinds of char sequences.

Methods

```
@SCJAllowed
@SCJMaySelfSuspend(false)
@SCJMayAllocate({
   javax.safetycritical.annotate.AllocationContext.CURRENT,
   javax.safetycritical.annotate.AllocationContext.INNER,
   javax.safetycritical.annotate.AllocationContext.OUTER})
@SCJPhase({
   javax.safetycritical.annotate.Phase.STARTUP,
   javax.safetycritical.annotate.Phase.INITIALIZATION,
   javax.safetycritical.annotate.Phase.RUN,
   javax.safetycritical.annotate.Phase.CLEANUP })
public char charAt(int index)
```

Implementations of this method must not allocate memory and must not allow ”this” to escape the local variables.

```
@SCJAllowed
@SCJMaySelfSuspend(false)
@SCJMayAllocate({
   javax.safetycritical.annotate.AllocationContext.CURRENT,
   javax.safetycritical.annotate.AllocationContext.INNER,
   javax.safetycritical.annotate.AllocationContext.OUTER})
@SCJPhase({
   javax.safetycritical.annotate.Phase.STARTUP,
   javax.safetycritical.annotate.Phase.INITIALIZATION,
   javax.safetycritical.annotate.Phase.RUN,
   javax.safetycritical.annotate.Phase.CLEANUP })
public int length( )
```

Implementations of this method must not allocate memory and must not allow ”this” to escape the local variables.

```
@SCJMayAllocate({javax.safetycritical.annotate.AllocationContext.CURRENT})
@SCJAllowed
@SCJMaySelfSuspend(false)
@SCJPhase({
   javax.safetycritical.annotate.Phase.STARTUP,
   javax.safetycritical.annotate.Phase.INITIALIZATION,
   javax.safetycritical.annotate.Phase.RUN,
   javax.safetycritical.annotate.Phase.CLEANUP })
public java.lang.CharSequence subSequence(int start, int end)
```

Implementations of this method may allocate a CharSequence object in the scope of the caller to hold the result of this method.

This method shall not allow ”this” to escape the local variables.

```
@SCJMayAllocate({javax.safetycritical.annotate.AllocationContext.CURRENT})
@SCJAllowed
@SCJMaySelfSuspend(false)
@SCJPhase({
   javax.safetycritical.annotate.Phase.STARTUP,
   javax.safetycritical.annotate.Phase.INITIALIZATION,
   javax.safetycritical.annotate.Phase.RUN,
   javax.safetycritical.annotate.Phase.CLEANUP })
public java.lang.String toString( )
```

Implementations of this method may allocate a String object in the scope of the caller to hold the result of this method.

This method shall not allow ”this” to escape the local variables.

B.2.3 INTERFACE Cloneable

```
@SCJAllowed
public interface Cloneable
```

A class implements the Cloneable interface to indicate to the Object.clone() method that it is legal for that method to make a field-for-field copy of instances of that class.

B.2.4 INTERFACE Comparable

```
@SCJAllowed
public interface Comparable<T>
```

This interface imposes a total ordering on the objects of each class that implements it. This ordering is referred to as the class's natural ordering, and the class's compareTo method is referred to as its natural comparison method.

Methods

```
@SCJAllowed
@SCJMaySelfSuspend(false)
@SCJMayAllocate({})
@SCJPhase({
```

```
    javax.safetycritical.annotate.Phase.STARTUP,
    javax.safetycritical.annotate.Phase.INITIALIZATION,
    javax.safetycritical.annotate.Phase.RUN,
    javax.safetycritical.annotate.Phase.CLEANUP })
public int compareTo(T o)
  throws java.lang.ClassCastException
```

The implementation of this method shall not allocate memory and shall not allow ”this” or ”o” argument to escape local variables.

o — The object to be compared.

returns a negative integer, zero, or a positive integer as this object is less than, equal to, or greater than the specified object.

Throws NullPointerException - if the specified object is null

Throws ClassCastException - if the specified object's type prevents it from being compared to this object.

B.2.5 INTERFACE **Runnable**

```
@SCJAllowed
public interface Runnable
```

The Runnable interface should be implemented by any class whose instances are intended to be executed by a thread. The class must define a method of no arguments called run.

Methods

```
@SCJAllowed
@SCJMayAllocate({
    javax.safetycritical.annotate.AllocationContext.CURRENT,
    javax.safetycritical.annotate.AllocationContext.INNER,
    javax.safetycritical.annotate.AllocationContext.OUTER})
@SCJMaySelfSuspend(true)
@SCJPhase({
    javax.safetycritical.annotate.Phase.STARTUP,
    javax.safetycritical.annotate.Phase.INITIALIZATION,
    javax.safetycritical.annotate.Phase.RUN,
    javax.safetycritical.annotate.Phase.CLEANUP })
public void run( )
```

The implementation of this method may, in general, perform allocations in immortal memory.

B.2.6 INTERFACE Thread.UncaughtExceptionHandler

```
@SCJAllowed(javax.safetycritical.annotate.Level.LEVEL_2)
public static interface Thread.UncaughtExceptionHandler
```

Interface for handlers invoked when a Thread abruptly terminates due to an uncaught exception.

Methods

```
@SCJAllowed(javax.safetycritical.annotate.Level.LEVEL_2)
@SCJMaySelfSuspend(false)
@SCJMayAllocate({})
@SCJPhase({
   javax.safetycritical.annotate.Phase.INITIALIZATION,
   javax.safetycritical.annotate.Phase.RUN,
   javax.safetycritical.annotate.Phase.CLEANUP})
public void uncaughtException(Thread t, Throwable e)
```

Establishes the interface for a handler for uncaught exceptions. Allocates no memory. Does not allow implicit argument this, or explicit arguments t and e to escape local variables.

B.2.7 INTERFACE UncaughtExceptionHandler

```
@SCJAllowed(javax.safetycritical.annotate.Level.LEVEL_2)
public interface UncaughtExceptionHandler
```

When a thread is about to terminate due to an uncaught exception, the SCJ implementation will query the thread for its UncaughtExceptionHandler using Thread.getUncaughtExceptionHandler() and will invoke the handler's uncaughtException method, passing the thread and the exception as arguments. If a thread has no special requirements for dealing with the exception, it can forward the invocation to the default uncaught exception handler.

Methods

```
@SCJAllowed
@SCJMaySelfSuspend(false)
@SCJMayAllocate({
   javax.safetycritical.annotate.AllocationContext.CURRENT,
   javax.safetycritical.annotate.AllocationContext.INNER,
   javax.safetycritical.annotate.AllocationContext.OUTER})
@SCJPhase({
```

```
    javax.safetycritical.annotate.Phase.STARTUP,
    javax.safetycritical.annotate.Phase.INITIALIZATION,
    javax.safetycritical.annotate.Phase.RUN,
    javax.safetycritical.annotate.Phase.CLEANUP })
public void uncaughtException(Thread t, Throwable e)
```

Method invoked when the given thread terminates due to the given uncaught exception.

Any exception thrown by this method will be ignored by the SCJ implementation.

t — the thread.

e — the exception.

B.3 Classes

B.3.1 CLASS ArithmeticException

```
@SCJAllowed
public class ArithmeticException implements java.io.Serializable extends
    java.lang.RuntimeException
```

Thrown when an exceptional arithmetic condition has occurred. For example, an integer ”divide by zero” throws an instance of this class.

Constructors

```
@SCJAllowed
@SCJMaySelfSuspend(false)
@SCJMayAllocate({javax.safetycritical.annotate.AllocationContext.CURRENT})
@SCJPhase({
    javax.safetycritical.annotate.Phase.STARTUP,
    javax.safetycritical.annotate.Phase.INITIALIZATION,
    javax.safetycritical.annotate.Phase.RUN,
    javax.safetycritical.annotate.Phase.CLEANUP })
public ArithmeticException( )
```

Shall not copy ”this” to any instance or static field.

Allocates an application- and implementation-dependent amount of memory in the current scope (to represent stack backtrace).

```
@SCJAllowed
@SCJMaySelfSuspend(false)
@SCJMayAllocate({javax.safetycritical.annotate.AllocationContext.CURRENT})
@SCJPhase({
   javax.safetycritical.annotate.Phase.STARTUP,
   javax.safetycritical.annotate.Phase.INITIALIZATION,
   javax.safetycritical.annotate.Phase.RUN,
   javax.safetycritical.annotate.Phase.CLEANUP })
public ArithmeticException(String msg)
```

Shall not copy ”this” to any instance or static field. The scope containing the msg argument must enclose the scope containing ”this”. Otherwise, an IllegalAssignmentError will be thrown.

Allocates an application- and implementation-dependent amount of memory in the current scope (to represent stack backtrace).

B.3.2 CLASS **ArrayIndexOutOfBoundsException**

```
@SCJAllowed
public class ArrayIndexOutOfBoundsException implements java.io.Serializable
      extends java.lang.IndexOutOfBoundsException
```

Thrown to indicate that an array has been accessed with an illegal index. The index is either negative or greater than or equal to the size of the array.

Constructors

```
@SCJAllowed
@SCJMaySelfSuspend(false)
@SCJMayAllocate({javax.safetycritical.annotate.AllocationContext.CURRENT})
@SCJPhase({
   javax.safetycritical.annotate.Phase.STARTUP,
   javax.safetycritical.annotate.Phase.INITIALIZATION,
   javax.safetycritical.annotate.Phase.RUN,
   javax.safetycritical.annotate.Phase.CLEANUP })
public ArrayIndexOutOfBoundsException( )
```

Shall not copy ”this” to any instance or static field.

Allocates an application- and implementation-dependent amount of memory in the current scope (to represent stack backtrace).

```
@SCJAllowed
@SCJMaySelfSuspend(false)
@SCJMayAllocate({javax.safetycritical.annotate.AllocationContext.CURRENT})
@SCJPhase({
   javax.safetycritical.annotate.Phase.STARTUP,
   javax.safetycritical.annotate.Phase.INITIALIZATION,
   javax.safetycritical.annotate.Phase.RUN,
   javax.safetycritical.annotate.Phase.CLEANUP })
public ArrayIndexOutOfBoundsException(int index)
```

Shall not copy ”this” to any instance or static field.

Allocates an application- and implementation-dependent amount of memory in the current scope (to represent stack backtrace).

```
@SCJAllowed
@SCJMaySelfSuspend(false)
@SCJMayAllocate({javax.safetycritical.annotate.AllocationContext.CURRENT})
@SCJPhase({
   javax.safetycritical.annotate.Phase.STARTUP,
   javax.safetycritical.annotate.Phase.INITIALIZATION,
   javax.safetycritical.annotate.Phase.RUN,
   javax.safetycritical.annotate.Phase.CLEANUP })
public ArrayIndexOutOfBoundsException(String msg)
```

Shall not copy ”this” to any instance or static field. The scope containing the msg argument must enclose the scope containing ”this”. Otherwise, an IllegalAssignmentError will be thrown.

Allocates an application- and implementation-dependent amount of memory in the current scope (to represent stack backtrace).

B.3.3 CLASS ArrayStoreException

```
@SCJAllowed
public class ArrayStoreException implements java.io.Serializable extends
    java.lang.RuntimeException
```

Thrown to indicate that an attempt has been made to store the wrong type of object into an array of objects.

Constructors

```
@SCJAllowed
@SCJMaySelfSuspend(false)
@SCJMayAllocate({javax.safetycritical.annotate.AllocationContext.CURRENT})
@SCJPhase({
   javax.safetycritical.annotate.Phase.STARTUP,
   javax.safetycritical.annotate.Phase.INITIALIZATION,
   javax.safetycritical.annotate.Phase.RUN,
   javax.safetycritical.annotate.Phase.CLEANUP })
public ArrayStoreException( )
```

Shall not copy ”this” to any instance or static field.

Allocates an application- and implementation-dependent amount of memory in the current scope (to represent stack backtrace).

```
@SCJAllowed
@SCJMaySelfSuspend(false)
@SCJMayAllocate({javax.safetycritical.annotate.AllocationContext.CURRENT})
@SCJPhase({
   javax.safetycritical.annotate.Phase.STARTUP,
   javax.safetycritical.annotate.Phase.INITIALIZATION,
   javax.safetycritical.annotate.Phase.RUN,
   javax.safetycritical.annotate.Phase.CLEANUP })
public ArrayStoreException(String msg)
```

Shall not copy ”this” to any instance or static field. The scope containing the msg argument must enclose the scope containing ”this”. Otherwise, an IllegalAssignmentError will be thrown.

Allocates an application- and implementation-dependent amount of memory in the current scope (to represent stack backtrace).

B.3.4 CLASS AssertionError

```
@SCJAllowed
public class AssertionError implements java.io.Serializable extends
    java.lang.Error
```

Thrown to indicate that an assertion has failed.

Constructors

```
@SCJAllowed
@SCJMaySelfSuspend(false)
@SCJMayAllocate({javax.safetycritical.annotate.AllocationContext.CURRENT})
```

```
@SCJPhase({
   javax.safetycritical.annotate.Phase.STARTUP,
   javax.safetycritical.annotate.Phase.INITIALIZATION,
   javax.safetycritical.annotate.Phase.RUN,
   javax.safetycritical.annotate.Phase.CLEANUP })
public AssertionError( )
```

Shall not copy ”this” to any instance or static field.

Allocates an application- and implementation-dependent amount of memory in the current scope (to represent stack backtrace).

```
@SCJAllowed
@SCJMaySelfSuspend(false)
@SCJMayAllocate({javax.safetycritical.annotate.AllocationContext.CURRENT})
@SCJPhase({
   javax.safetycritical.annotate.Phase.STARTUP,
   javax.safetycritical.annotate.Phase.INITIALIZATION,
   javax.safetycritical.annotate.Phase.RUN,
   javax.safetycritical.annotate.Phase.CLEANUP })
public AssertionError(boolean b)
```

Shall not copy ”this” to any instance or static field.

Allocates an application- and implementation-dependent amount of memory in the current scope (to represent stack backtrace).

```
@SCJAllowed
@SCJMaySelfSuspend(false)
@SCJMayAllocate({javax.safetycritical.annotate.AllocationContext.CURRENT})
@SCJPhase({
   javax.safetycritical.annotate.Phase.STARTUP,
   javax.safetycritical.annotate.Phase.INITIALIZATION,
   javax.safetycritical.annotate.Phase.RUN,
   javax.safetycritical.annotate.Phase.CLEANUP })
public AssertionError(char c)
```

Shall not copy ”this” to any instance or static field.

Allocates an application- and implementation-dependent amount of memory in the current scope (to represent stack backtrace).

```
@SCJAllowed
@SCJMaySelfSuspend(false)
@SCJMayAllocate({javax.safetycritical.annotate.AllocationContext.CURRENT})
@SCJPhase({
```

```
    javax.safetycritical.annotate.Phase.STARTUP,
    javax.safetycritical.annotate.Phase.INITIALIZATION,
    javax.safetycritical.annotate.Phase.RUN,
    javax.safetycritical.annotate.Phase.CLEANUP })
public AssertionError(double d)
```

Shall not copy ”this” to any instance or static field.

Allocates an application- and implementation-dependent amount of memory in the current scope (to represent stack backtrace).

```
@SCJAllowed
@SCJMaySelfSuspend(false)
@SCJMayAllocate({javax.safetycritical.annotate.AllocationContext.CURRENT})
@SCJPhase({
    javax.safetycritical.annotate.Phase.STARTUP,
    javax.safetycritical.annotate.Phase.INITIALIZATION,
    javax.safetycritical.annotate.Phase.RUN,
    javax.safetycritical.annotate.Phase.CLEANUP })
public AssertionError(float f)
```

Shall not copy ”this” to any instance or static field.

Allocates an application- and implementation-dependent amount of memory in the current scope (to represent stack backtrace).

```
@SCJAllowed
@SCJMaySelfSuspend(false)
@SCJMayAllocate({javax.safetycritical.annotate.AllocationContext.CURRENT})
@SCJPhase({
    javax.safetycritical.annotate.Phase.STARTUP,
    javax.safetycritical.annotate.Phase.INITIALIZATION,
    javax.safetycritical.annotate.Phase.RUN,
    javax.safetycritical.annotate.Phase.CLEANUP })
public AssertionError(int i)
```

Shall not copy ”this” to any instance or static field.

Allocates an application- and implementation-dependent amount of memory in the current scope (to represent stack backtrace).

```
@SCJAllowed
@SCJMaySelfSuspend(false)
@SCJMayAllocate({javax.safetycritical.annotate.AllocationContext.CURRENT})
@SCJPhase({
    javax.safetycritical.annotate.Phase.STARTUP,
```

```
    javax.safetycritical.annotate.Phase.INITIALIZATION,
    javax.safetycritical.annotate.Phase.RUN,
    javax.safetycritical.annotate.Phase.CLEANUP })
public AssertionError(long l)
```

Shall not copy ”this” to any instance or static field.

Allocates an application- and implementation-dependent amount of memory in the current scope (to represent stack backtrace).

```
@SCJAllowed
@SCJMaySelfSuspend(false)
@SCJMayAllocate({javax.safetycritical.annotate.AllocationContext.CURRENT})
@SCJPhase({
    javax.safetycritical.annotate.Phase.STARTUP,
    javax.safetycritical.annotate.Phase.INITIALIZATION,
    javax.safetycritical.annotate.Phase.RUN,
    javax.safetycritical.annotate.Phase.CLEANUP })
public AssertionError(Object o)
```

Shall not copy ”this” to any instance or static field.

Allocates an application- and implementation-dependent amount of memory in the current scope (to represent stack backtrace).

B.3.5 CLASS **Boolean**

```
@SCJAllowed
public class Boolean implements java.lang.Comparable<T>, java.io.Serializable
    extends java.lang.Object
```

The Boolean class wraps a value of the primitive type boolean in an object. An object of type Boolean contains a single field whose type is boolean.

Fields

```
@SCJAllowed
public static final java.lang.Boolean FALSE
```

```
@SCJAllowed
public static final java.lang.Boolean TRUE
```

```
@SCJAllowed
public static final java.lang.Class<java.lang.Boolean> TYPE
```

Constructors

```
@SCJAllowed
@SCJMaySelfSuspend(false)
@SCJMayAllocate({})
@SCJPhase({
    javax.safetycritical.annotate.Phase.STARTUP,
    javax.safetycritical.annotate.Phase.INITIALIZATION,
    javax.safetycritical.annotate.Phase.RUN,
    javax.safetycritical.annotate.Phase.CLEANUP })
public Boolean(boolean v)
```

Allocates no memory. Does not allow ”this” to escape local variables.

```
@SCJAllowed
@SCJMaySelfSuspend(false)
@SCJMayAllocate({})
@SCJPhase({
    javax.safetycritical.annotate.Phase.STARTUP,
    javax.safetycritical.annotate.Phase.INITIALIZATION,
    javax.safetycritical.annotate.Phase.RUN,
    javax.safetycritical.annotate.Phase.CLEANUP })
public Boolean(String str)
```

Allocates no memory. Does not allow ”this” or ”str” argument to escape local variables.

Methods

```
@SCJAllowed
@SCJMaySelfSuspend(false)
@SCJMayAllocate({})
@SCJPhase({
    javax.safetycritical.annotate.Phase.STARTUP,
    javax.safetycritical.annotate.Phase.INITIALIZATION,
    javax.safetycritical.annotate.Phase.RUN,
    javax.safetycritical.annotate.Phase.CLEANUP })
public boolean booleanValue( )
```

Allocates no memory. Does not allow ”this” to escape local variables.

```
@SCJAllowed
@SCJMaySelfSuspend(false)
@SCJMayAllocate({})
@SCJPhase({
```

```
    javax.safetycritical.annotate.Phase.STARTUP,
    javax.safetycritical.annotate.Phase.INITIALIZATION,
    javax.safetycritical.annotate.Phase.RUN,
    javax.safetycritical.annotate.Phase.CLEANUP })
public int compareTo(Boolean b)
```

Allocates no memory. Does not allow ”this” or argument ”b” to escape local variables.

```
@Override
@SCJAllowed
@SCJMaySelfSuspend(false)
@SCJMayAllocate({})
@SCJPhase({
    javax.safetycritical.annotate.Phase.STARTUP,
    javax.safetycritical.annotate.Phase.INITIALIZATION,
    javax.safetycritical.annotate.Phase.RUN,
    javax.safetycritical.annotate.Phase.CLEANUP })
public boolean equals(Object obj)
```

Allocates no memory. Does not allow ”this” or argument ”obj” to escape local variables.

```
@SCJAllowed
@SCJMaySelfSuspend(false)
@SCJMayAllocate({})
@SCJPhase({
    javax.safetycritical.annotate.Phase.STARTUP,
    javax.safetycritical.annotate.Phase.INITIALIZATION,
    javax.safetycritical.annotate.Phase.RUN,
    javax.safetycritical.annotate.Phase.CLEANUP })
public static boolean getBoolean(String str)
```

Allocates no memory. Does not allow argument ”str” to escape local variables.

```
@Override
@SCJAllowed
@SCJMaySelfSuspend(false)
@SCJMayAllocate({})
@SCJPhase({
    javax.safetycritical.annotate.Phase.STARTUP,
    javax.safetycritical.annotate.Phase.INITIALIZATION,
    javax.safetycritical.annotate.Phase.RUN,
    javax.safetycritical.annotate.Phase.CLEANUP })
public int hashCode( )
```

Allocates no memory. Does not allow ”this” to escape local variables.

```
@SCJAllowed
@SCJMaySelfSuspend(false)
@SCJMayAllocate({})
@SCJPhase({
   javax.safetycritical.annotate.Phase.STARTUP,
   javax.safetycritical.annotate.Phase.INITIALIZATION,
   javax.safetycritical.annotate.Phase.RUN,
   javax.safetycritical.annotate.Phase.CLEANUP })
public static boolean parseBoolean(String str)
```

Allocates no memory. Does not allow argument ”str” to escape local variables.

```
@SCJAllowed
@SCJMaySelfSuspend(false)
@SCJMayAllocate({})
@SCJPhase({
   javax.safetycritical.annotate.Phase.STARTUP,
   javax.safetycritical.annotate.Phase.INITIALIZATION,
   javax.safetycritical.annotate.Phase.RUN,
   javax.safetycritical.annotate.Phase.CLEANUP })
public static java.lang.String toString(boolean value)
```

Allocates no memory. Returns a String literal which resides in the scope of the Classloader that is responsible for loading the Boolean class.

```
@Override
@SCJAllowed
@SCJMaySelfSuspend(false)
@SCJMayAllocate({javax.safetycritical.annotate.AllocationContext.CURRENT})
@SCJPhase({
   javax.safetycritical.annotate.Phase.STARTUP,
   javax.safetycritical.annotate.Phase.INITIALIZATION,
   javax.safetycritical.annotate.Phase.RUN,
   javax.safetycritical.annotate.Phase.CLEANUP })
public java.lang.String toString( )
```

Does not allow ”this” to escape local variables. Allocates a String and associated internal ”structure” (e.g. char[]) in caller’s scope. (Note: this semantics is desired for consistency with overridden implementation of Object.toString()).

```
@SCJAllowed
@SCJMaySelfSuspend(false)
@SCJMayAllocate({})
@SCJPhase({
   javax.safetycritical.annotate.Phase.STARTUP,
   javax.safetycritical.annotate.Phase.INITIALIZATION,
   javax.safetycritical.annotate.Phase.RUN,
   javax.safetycritical.annotate.Phase.CLEANUP })
public static java.lang.Boolean valueOf(boolean b)
```

Allocates no memory. Returns a Boolean literal which resides at the scope of the Classloader that is responsible for loading the Boolean class.

```
@SCJAllowed
@SCJMaySelfSuspend(false)
@SCJMayAllocate({})
@SCJPhase({
   javax.safetycritical.annotate.Phase.STARTUP,
   javax.safetycritical.annotate.Phase.INITIALIZATION,
   javax.safetycritical.annotate.Phase.RUN,
   javax.safetycritical.annotate.Phase.CLEANUP })
public static java.lang.Boolean valueOf(String str)
```

Allocates no memory. Does not allow argument ”str” to escale local variables. Returns a Boolean literal which resides at the scope of the Classloader that is responsible for loading the Boolean class.

B.3.6 CLASS Byte

```
@SCJAllowed
public class Byte implements java.lang.Comparable<T>, java.io.Serializable
    extends java.lang.Number
```

The Byte class wraps a value of primitive type byte in an object. An object of type Byte contains a single field whose type is byte.

Fields

```
@SCJAllowed
public static final byte MAX_VALUE
```

```
@SCJAllowed
public static final byte MIN_VALUE
```

```
@SCJAllowed
public static final int SIZE
```

```
@SCJAllowed
public static final java.lang.Class<java.lang.Byte> TYPE
```

Constructors

```
@SCJAllowed
@SCJMaySelfSuspend(false)
@SCJMayAllocate({})
@SCJPhase({
    javax.safetycritical.annotate.Phase.STARTUP,
    javax.safetycritical.annotate.Phase.INITIALIZATION,
    javax.safetycritical.annotate.Phase.RUN,
    javax.safetycritical.annotate.Phase.CLEANUP })
public Byte(byte val)
```

Allocates no memory. Does not allow ”this” to escape local variables.

```
@SCJAllowed
@SCJMaySelfSuspend(false)
@SCJMayAllocate({})
@SCJPhase({
    javax.safetycritical.annotate.Phase.STARTUP,
    javax.safetycritical.annotate.Phase.INITIALIZATION,
    javax.safetycritical.annotate.Phase.RUN,
    javax.safetycritical.annotate.Phase.CLEANUP })
public Byte(String str)
  throws java.lang.NumberFormatException
```

Allocates no memory. Does not allow ”this” or ”str” argument to escape local variables.

Methods

```
@SCJAllowed
@SCJMaySelfSuspend(false)
@SCJMayAllocate({})
@SCJPhase({
    javax.safetycritical.annotate.Phase.STARTUP,
    javax.safetycritical.annotate.Phase.INITIALIZATION,
    javax.safetycritical.annotate.Phase.RUN,
    javax.safetycritical.annotate.Phase.CLEANUP })
public byte byteValue( )
```

Allocates no memory. Does not allow ”this” to escape local variables.

```
@SCJAllowed
@SCJMaySelfSuspend(false)
@SCJMayAllocate({})
@SCJPhase({
   javax.safetycritical.annotate.Phase.STARTUP,
   javax.safetycritical.annotate.Phase.INITIALIZATION,
   javax.safetycritical.annotate.Phase.RUN,
   javax.safetycritical.annotate.Phase.CLEANUP })
public int compareTo(Byte other)
```

Allocates no memory. Does not allow ”this” or ”other” argument to escape local variables.

```
@SCJAllowed
@SCJMaySelfSuspend(false)
@SCJMayAllocate({})
@SCJPhase({
   javax.safetycritical.annotate.Phase.STARTUP,
   javax.safetycritical.annotate.Phase.INITIALIZATION,
   javax.safetycritical.annotate.Phase.RUN,
   javax.safetycritical.annotate.Phase.CLEANUP })
public static java.lang.Byte decode(String str)
  throws java.lang.NumberFormatException
```

Does not allow ”str” argument to escape local variables. Allocates a Byte result object in the caller’s scope.

```
@SCJAllowed
@SCJMaySelfSuspend(false)
@SCJMayAllocate({})
@SCJPhase({
   javax.safetycritical.annotate.Phase.STARTUP,
   javax.safetycritical.annotate.Phase.INITIALIZATION,
   javax.safetycritical.annotate.Phase.RUN,
   javax.safetycritical.annotate.Phase.CLEANUP })
public double doubleValue( )
```

Allocates no memory. Does not allow ”this” to escape local variables.

```
@SCJAllowed
@SCJMaySelfSuspend(false)
@SCJMayAllocate({})
```

```
@SCJPhase({
    javax.safetycritical.annotate.Phase.STARTUP,
    javax.safetycritical.annotate.Phase.INITIALIZATION,
    javax.safetycritical.annotate.Phase.RUN,
    javax.safetycritical.annotate.Phase.CLEANUP })
public boolean equals(Object obj)
```

Allocates no memory. Does not allow ”this” or ”obj” argument to escape local variables.

```
@SCJAllowed
@SCJMaySelfSuspend(false)
@SCJMayAllocate({})
@SCJPhase({
    javax.safetycritical.annotate.Phase.STARTUP,
    javax.safetycritical.annotate.Phase.INITIALIZATION,
    javax.safetycritical.annotate.Phase.RUN,
    javax.safetycritical.annotate.Phase.CLEANUP })
public float floatValue( )
```

Allocates no memory. Does not allow ”this” to escape local variables.

```
@SCJAllowed
@SCJMaySelfSuspend(false)
@SCJMayAllocate({})
@SCJPhase({
    javax.safetycritical.annotate.Phase.STARTUP,
    javax.safetycritical.annotate.Phase.INITIALIZATION,
    javax.safetycritical.annotate.Phase.RUN,
    javax.safetycritical.annotate.Phase.CLEANUP })
public int hashCode( )
```

Allocates no memory. Does not allow ”this” to escape local variables.

```
@SCJAllowed
@SCJMaySelfSuspend(false)
@SCJMayAllocate({})
@SCJPhase({
    javax.safetycritical.annotate.Phase.STARTUP,
    javax.safetycritical.annotate.Phase.INITIALIZATION,
    javax.safetycritical.annotate.Phase.RUN,
    javax.safetycritical.annotate.Phase.CLEANUP })
public int intValue( )
```

Allocates no memory. Does not allow ”this” to escape local variables.

```
@SCJAllowed
@SCJMaySelfSuspend(false)
@SCJMayAllocate({})
@SCJPhase({
    javax.safetycritical.annotate.Phase.STARTUP,
    javax.safetycritical.annotate.Phase.INITIALIZATION,
    javax.safetycritical.annotate.Phase.RUN,
    javax.safetycritical.annotate.Phase.CLEANUP })
public long longValue( )
```

Allocates no memory. Does not allow ”this” to escape local variables.

```
@SCJAllowed
@SCJMaySelfSuspend(false)
@SCJMayAllocate({})
@SCJPhase({
    javax.safetycritical.annotate.Phase.STARTUP,
    javax.safetycritical.annotate.Phase.INITIALIZATION,
    javax.safetycritical.annotate.Phase.RUN,
    javax.safetycritical.annotate.Phase.CLEANUP })
public static byte parseByte(String str, int base)
  throws java.lang.NumberFormatException
```

Allocates no memory. Does not allow ”str” argument to escape local variables.

```
@SCJAllowed
@SCJMaySelfSuspend(false)
@SCJMayAllocate({})
@SCJPhase({
    javax.safetycritical.annotate.Phase.STARTUP,
    javax.safetycritical.annotate.Phase.INITIALIZATION,
    javax.safetycritical.annotate.Phase.RUN,
    javax.safetycritical.annotate.Phase.CLEANUP })
public static byte parseByte(String str)
  throws java.lang.NumberFormatException
```

Allocates no memory. Does not allow ”str” argument to escape local variables.

```
@SCJAllowed
@SCJMaySelfSuspend(false)
@SCJMayAllocate({})
@SCJPhase({
    javax.safetycritical.annotate.Phase.STARTUP,
    javax.safetycritical.annotate.Phase.INITIALIZATION,
    javax.safetycritical.annotate.Phase.RUN,
    javax.safetycritical.annotate.Phase.CLEANUP })
public short shortValue( )
```

Allocates no memory. Does not allow ”this” to escape local variables.

```
@SCJAllowed
@SCJMaySelfSuspend(false)
@SCJMayAllocate({javax.safetycritical.annotate.AllocationContext.CURRENT})
@SCJPhase({
   javax.safetycritical.annotate.Phase.STARTUP,
   javax.safetycritical.annotate.Phase.INITIALIZATION,
   javax.safetycritical.annotate.Phase.RUN,
   javax.safetycritical.annotate.Phase.CLEANUP })
public static java.lang.String toString(byte v)
```

Allocates a String and associated internal ”structure” (e.g. char[]) in caller’s scope.

```
@SCJAllowed
@SCJMaySelfSuspend(false)
@SCJMayAllocate({javax.safetycritical.annotate.AllocationContext.CURRENT})
@SCJPhase({
   javax.safetycritical.annotate.Phase.STARTUP,
   javax.safetycritical.annotate.Phase.INITIALIZATION,
   javax.safetycritical.annotate.Phase.RUN,
   javax.safetycritical.annotate.Phase.CLEANUP })
public java.lang.String toString( )
```

Does not allow ”this” to escape local variables. Allocates a String and associated internal ”structure” (e.g. char[]) in caller’s scope. (Note: this semantics is desired for consistency with overridden implementation of Object.toString()).

```
@SCJAllowed
@SCJMaySelfSuspend(false)
@SCJMayAllocate({javax.safetycritical.annotate.AllocationContext.CURRENT})
@SCJPhase({
   javax.safetycritical.annotate.Phase.STARTUP,
   javax.safetycritical.annotate.Phase.INITIALIZATION,
   javax.safetycritical.annotate.Phase.RUN,
   javax.safetycritical.annotate.Phase.CLEANUP })
public static java.lang.Byte valueOf(String str, int base)
  throws java.lang.NumberFormatException
```

Does not allow ”str” argument to escape local variables. Allocates one Byte object in the caller’s scope.

```
@SCJAllowed
@SCJMaySelfSuspend(false)
@SCJMayAllocate({javax.safetycritical.annotate.AllocationContext.CURRENT})
@SCJPhase({
   javax.safetycritical.annotate.Phase.STARTUP,
   javax.safetycritical.annotate.Phase.INITIALIZATION,
   javax.safetycritical.annotate.Phase.RUN,
   javax.safetycritical.annotate.Phase.CLEANUP })
public static java.lang.Byte valueOf(byte val)
```

Allocates one Byte object in the caller's scope.

```
@SCJAllowed
@SCJMaySelfSuspend(false)
@SCJMayAllocate({javax.safetycritical.annotate.AllocationContext.CURRENT})
@SCJPhase({
   javax.safetycritical.annotate.Phase.STARTUP,
   javax.safetycritical.annotate.Phase.INITIALIZATION,
   javax.safetycritical.annotate.Phase.RUN,
   javax.safetycritical.annotate.Phase.CLEANUP })
public static java.lang.Byte valueOf(String str)
  throws java.lang.NumberFormatException
```

Does not allow "str" argument to escape local variables. Allocates one Byte object in the caller's scope.

B.3.7 CLASS Character

```
@SCJAllowed
public final class Character implements java.lang.Comparable<T>,
    java.io.Serializable extends java.lang.Object
```

The Character class wraps a value of the primitive type char in an object. An object of type Character contains a single field whose type is char.

Fields

```
@SCJAllowed
public static final byte COMBINING_SPACING_MARK
```

```
@SCJAllowed
public static final byte CONNECTOR_PUNCTUATION
```

@SCJAllowed
public static final byte CONTROL

@SCJAllowed
public static final byte CURRENCY_SYMBOL

@SCJAllowed
public static final byte DASH_PUNCTUATION

@SCJAllowed
public static final byte DECIMAL_DIGIT_NUMBER

@SCJAllowed
public static final byte ENCLOSING_MARK

@SCJAllowed
public static final byte END_PUNCTUATION

@SCJAllowed
public static final byte FINAL_QUOTE_PUNCTUATION

@SCJAllowed
public static final byte FORMAT

@SCJAllowed
public static final byte INITIAL_QUOTE_PUNCTUATION

@SCJAllowed
public static final byte LETTER_NUMBER

@SCJAllowed
public static final byte LINE_SEPARATOR

@SCJAllowed
public static final byte LOWERCASE_LETTER

@SCJAllowed
public static final byte MATH_SYMBOL

@SCJAllowed
public static final int MAX_RADIX

@SCJAllowed
public static final char MAX_VALUE

@SCJAllowed
public static final int MIN_RADIX

@SCJAllowed
public static final char MIN_VALUE

@SCJAllowed
public static final byte MODIFIER_LETTER

@SCJAllowed
public static final byte MODIFIER_SYMBOL

@SCJAllowed
public static final byte NON_SPACING_MARK

@SCJAllowed
public static final byte OTHER_LETTER

@SCJAllowed
public static final byte OTHER_NUMBER

@SCJAllowed
public static final byte OTHER_PUNCTUATION

@SCJAllowed
public static final byte OTHER_SYMBOL

@SCJAllowed
public static final byte PARAGRAPH_SEPARATOR

@SCJAllowed
public static final byte PRIVATE_USE

@SCJAllowed
public static final int SIZE

@SCJAllowed
public static final byte SPACE_SEPARATOR

@SCJAllowed
public static final byte START_PUNCTUATION

@SCJAllowed
public static final byte SURROGATE

@SCJAllowed
public static final byte TITLECASE_LETTER

@SCJAllowed
public static final java.lang.**Class**<java.lang.Character> TYPE

@SCJAllowed
public static final byte UNASSIGNED

@SCJAllowed
public static final byte UPPERCASE_LETTER

Constructors

```
@SCJAllowed
@SCJMaySelfSuspend(false)
@SCJMayAllocate({})
@SCJPhase({
    javax.safetycritical.annotate.Phase.STARTUP,
    javax.safetycritical.annotate.Phase.INITIALIZATION,
    javax.safetycritical.annotate.Phase.RUN,
    javax.safetycritical.annotate.Phase.CLEANUP })
public Character(char v)
```

Allocates no memory. Does not allow ”this” to escape local variables.

Methods

```
@SCJAllowed
@SCJMaySelfSuspend(false)
@SCJMayAllocate({})
@SCJPhase({
    javax.safetycritical.annotate.Phase.STARTUP,
    javax.safetycritical.annotate.Phase.INITIALIZATION,
    javax.safetycritical.annotate.Phase.RUN,
    javax.safetycritical.annotate.Phase.CLEANUP })
public char charValue( )
```

Allocates no memory. Does not allow ”this” to escape local variables.

```
@SCJAllowed
@SCJMaySelfSuspend(false)
@SCJMayAllocate({})
@SCJPhase({
   javax.safetycritical.annotate.Phase.STARTUP,
   javax.safetycritical.annotate.Phase.INITIALIZATION,
   javax.safetycritical.annotate.Phase.RUN,
   javax.safetycritical.annotate.Phase.CLEANUP })
public int compareTo(Character another_character)
```

Allocates no memory. Does not allow ”this” or ”another_character” argument to escape local variables.

```
@SCJAllowed
@SCJMaySelfSuspend(false)
@SCJMayAllocate({})
@SCJPhase({
   javax.safetycritical.annotate.Phase.STARTUP,
   javax.safetycritical.annotate.Phase.INITIALIZATION,
   javax.safetycritical.annotate.Phase.RUN,
   javax.safetycritical.annotate.Phase.CLEANUP })
public static int digit(char ch, int radix)
```

Allocates no memory.

```
@SCJAllowed
@SCJMaySelfSuspend(false)
@SCJMayAllocate({})
@SCJPhase({
   javax.safetycritical.annotate.Phase.STARTUP,
   javax.safetycritical.annotate.Phase.INITIALIZATION,
   javax.safetycritical.annotate.Phase.RUN,
   javax.safetycritical.annotate.Phase.CLEANUP })
public boolean equals(Object obj)
```

Allocates no memory. Does not allow ”this” or ”obj” argument to escape local variables.

```
@SCJAllowed
@SCJMaySelfSuspend(false)
@SCJMayAllocate({})
@SCJPhase({
   javax.safetycritical.annotate.Phase.STARTUP,
   javax.safetycritical.annotate.Phase.INITIALIZATION,
   javax.safetycritical.annotate.Phase.RUN,
   javax.safetycritical.annotate.Phase.CLEANUP })
public static int getType(char ch)
```

Allocates no memory.

```
@SCJAllowed
@SCJMaySelfSuspend(false)
@SCJMayAllocate({})
@SCJPhase({
    javax.safetycritical.annotate.Phase.STARTUP,
    javax.safetycritical.annotate.Phase.INITIALIZATION,
    javax.safetycritical.annotate.Phase.RUN,
    javax.safetycritical.annotate.Phase.CLEANUP })
public int hashCode( )
```

```
@SCJAllowed
@SCJMaySelfSuspend(false)
@SCJMayAllocate({})
@SCJPhase({
    javax.safetycritical.annotate.Phase.STARTUP,
    javax.safetycritical.annotate.Phase.INITIALIZATION,
    javax.safetycritical.annotate.Phase.RUN,
    javax.safetycritical.annotate.Phase.CLEANUP })
public static boolean isLetter(char ch)
```

Allocates no memory.

```
@SCJAllowed
@SCJMaySelfSuspend(false)
@SCJMayAllocate({})
@SCJPhase({
    javax.safetycritical.annotate.Phase.STARTUP,
    javax.safetycritical.annotate.Phase.INITIALIZATION,
    javax.safetycritical.annotate.Phase.RUN,
    javax.safetycritical.annotate.Phase.CLEANUP })
public static boolean isLetterOrDigit(char ch)
```

Allocates no memory.

```
@SCJAllowed
@SCJMaySelfSuspend(false)
@SCJMayAllocate({})
@SCJPhase({
    javax.safetycritical.annotate.Phase.STARTUP,
    javax.safetycritical.annotate.Phase.INITIALIZATION,
    javax.safetycritical.annotate.Phase.RUN,
    javax.safetycritical.annotate.Phase.CLEANUP })
public static boolean isLowerCase(char ch)
```

Allocates no memory.

```
@SCJAllowed
@SCJMaySelfSuspend(false)
@SCJMayAllocate({})
@SCJPhase({
    javax.safetycritical.annotate.Phase.STARTUP,
    javax.safetycritical.annotate.Phase.INITIALIZATION,
    javax.safetycritical.annotate.Phase.RUN,
    javax.safetycritical.annotate.Phase.CLEANUP })
public static boolean isSpaceChar(char ch)
```

Allocates no memory.

```
@SCJAllowed
@SCJMaySelfSuspend(false)
@SCJMayAllocate({})
@SCJPhase({
    javax.safetycritical.annotate.Phase.STARTUP,
    javax.safetycritical.annotate.Phase.INITIALIZATION,
    javax.safetycritical.annotate.Phase.RUN,
    javax.safetycritical.annotate.Phase.CLEANUP })
public static boolean isUpperCase(char ch)
```

Allocates no memory.

```
@SCJAllowed
@SCJMaySelfSuspend(false)
@SCJMayAllocate({})
@SCJPhase({
    javax.safetycritical.annotate.Phase.STARTUP,
    javax.safetycritical.annotate.Phase.INITIALIZATION,
    javax.safetycritical.annotate.Phase.RUN,
    javax.safetycritical.annotate.Phase.CLEANUP })
public static boolean isWhitespace(char ch)
```

Allocates no memory.

```
@SCJAllowed
@SCJMaySelfSuspend(false)
@SCJMayAllocate({})
@SCJPhase({
    javax.safetycritical.annotate.Phase.STARTUP,
    javax.safetycritical.annotate.Phase.INITIALIZATION,
    javax.safetycritical.annotate.Phase.RUN,
    javax.safetycritical.annotate.Phase.CLEANUP })
public static char toLowerCase(char ch)
```

Allocates no memory.

```
@SCJAllowed
@SCJMaySelfSuspend(false)
@SCJMayAllocate({javax.safetycritical.annotate.AllocationContext.CURRENT})
@SCJPhase({
   javax.safetycritical.annotate.Phase.STARTUP,
   javax.safetycritical.annotate.Phase.INITIALIZATION,
   javax.safetycritical.annotate.Phase.RUN,
   javax.safetycritical.annotate.Phase.CLEANUP })
public static java.lang.String toString(char c)
```

Allocates a String and associated internal ”structure” (e.g. char[]) in caller’s scope.

```
@SCJAllowed
@SCJMaySelfSuspend(false)
@SCJMayAllocate({javax.safetycritical.annotate.AllocationContext.CURRENT})
@SCJPhase({
   javax.safetycritical.annotate.Phase.STARTUP,
   javax.safetycritical.annotate.Phase.INITIALIZATION,
   javax.safetycritical.annotate.Phase.RUN,
   javax.safetycritical.annotate.Phase.CLEANUP })
public java.lang.String toString( )
```

Does not allow ”this” to escape local variables. Allocates a String and associated internal ”structure” (e.g. char[]) in caller’s scope. (Note: this semantics is desired for consistency with overridden implementation of Object.toString()).

```
@SCJAllowed
@SCJMaySelfSuspend(false)
@SCJMayAllocate({})
@SCJPhase({
   javax.safetycritical.annotate.Phase.STARTUP,
   javax.safetycritical.annotate.Phase.INITIALIZATION,
   javax.safetycritical.annotate.Phase.RUN,
   javax.safetycritical.annotate.Phase.CLEANUP })
public static char toUpperCase(char ch)
```

Allocates no memory.

```
@SCJAllowed
@SCJMaySelfSuspend(false)
@SCJMayAllocate({javax.safetycritical.annotate.AllocationContext.CURRENT})
```

```
@SCJPhase({
    javax.safetycritical.annotate.Phase.STARTUP,
    javax.safetycritical.annotate.Phase.INITIALIZATION,
    javax.safetycritical.annotate.Phase.RUN,
    javax.safetycritical.annotate.Phase.CLEANUP })
public static java.lang.Character valueOf(char c)
```

Allocates a Character object in caller's scope.

B.3.8 CLASS Class

```
@SCJAllowed
public final class Class<T> implements java.io.Serializable extends
    java.lang.Object
```

Instances of the class Class represent classes and interfaces in a running Java application.

Methods

```
@SCJAllowed
@SCJMaySelfSuspend(false)
@SCJMayAllocate({})
@SCJPhase({
    javax.safetycritical.annotate.Phase.STARTUP,
    javax.safetycritical.annotate.Phase.INITIALIZATION,
    javax.safetycritical.annotate.Phase.RUN,
    javax.safetycritical.annotate.Phase.CLEANUP })
public boolean desiredAssertionStatus( )
```

Allocates no memory. Does not allow "this" to escape local variables.

```
@SCJAllowed
@SCJMaySelfSuspend(false)
@SCJMayAllocate({})
@SCJPhase({
    javax.safetycritical.annotate.Phase.STARTUP,
    javax.safetycritical.annotate.Phase.INITIALIZATION,
    javax.safetycritical.annotate.Phase.RUN,
    javax.safetycritical.annotate.Phase.CLEANUP })
public java.lang.Class<?> getComponentType( )
```

Allocates no memory. Does not allow "this" to escape local variables.

Returns a reference to a previously allocated Class object, which resides in the scope of its ClassLoader.

```
@SCJAllowed
@SCJMaySelfSuspend(false)
@SCJMayAllocate({})
@SCJPhase({
   javax.safetycritical.annotate.Phase.STARTUP,
   javax.safetycritical.annotate.Phase.INITIALIZATION,
   javax.safetycritical.annotate.Phase.RUN,
   javax.safetycritical.annotate.Phase.CLEANUP })
public java.lang.Class<?> getDeclaringClass( )
```

Allocates no memory. Returns a reference to a previously existing Class, which resides in the scope of its ClassLoader.

```
@SCJAllowed
@SCJMaySelfSuspend(false)
@SCJMayAllocate({javax.safetycritical.annotate.AllocationContext.CURRENT})
@SCJPhase({
   javax.safetycritical.annotate.Phase.STARTUP,
   javax.safetycritical.annotate.Phase.INITIALIZATION,
   javax.safetycritical.annotate.Phase.RUN,
   javax.safetycritical.annotate.Phase.CLEANUP })
public <T[]> T[] getEnumConstants( )
```

Does not alow ”this” to escape local variables.

Allocates an array of T in the caller's scope. The allocated array holds references to previously allocated T objects. Thus, the existing T objects must reside in a scope that encloses the caller's scope. Note that the existing T objects reside in the scope of the corresponding ClassLoader.

```
@SCJAllowed
@SCJMaySelfSuspend(false)
@SCJMayAllocate({})
@SCJPhase({
   javax.safetycritical.annotate.Phase.STARTUP,
   javax.safetycritical.annotate.Phase.INITIALIZATION,
   javax.safetycritical.annotate.Phase.RUN,
   javax.safetycritical.annotate.Phase.CLEANUP })
public java.lang.String getName( )
```

Allocates no memory. Does not allow ”this” to escape local variables.

Returns a reference to a previously allocated String object, which resides in the scope of this Class's ClassLoader or in some enclosing scope.

```
@SCJAllowed
@SCJMaySelfSuspend(false)
@SCJMayAllocate({})
@SCJPhase({
   javax.safetycritical.annotate.Phase.STARTUP,
   javax.safetycritical.annotate.Phase.INITIALIZATION,
   javax.safetycritical.annotate.Phase.RUN,
   javax.safetycritical.annotate.Phase.CLEANUP })
public java.lang.Class<? super T> getSuperclass( )
```

Allocates no memory. Does not allow ”this” to escape local variables.

Returns a reference to a previously allocated Class object, which resides in the scope of its ClassLoader.

```
@SCJAllowed
@SCJMaySelfSuspend(false)
@SCJMayAllocate({})
@SCJPhase({
   javax.safetycritical.annotate.Phase.STARTUP,
   javax.safetycritical.annotate.Phase.INITIALIZATION,
   javax.safetycritical.annotate.Phase.RUN,
   javax.safetycritical.annotate.Phase.CLEANUP })
public boolean isAnnotation( )
```

Allocates no memory. Does not allow ”this” to escape local variables.

```
@SCJAllowed
@SCJMaySelfSuspend(false)
@SCJMayAllocate({})
@SCJPhase({
   javax.safetycritical.annotate.Phase.STARTUP,
   javax.safetycritical.annotate.Phase.INITIALIZATION,
   javax.safetycritical.annotate.Phase.RUN,
   javax.safetycritical.annotate.Phase.CLEANUP })
public boolean isArray( )
```

Allocates no memory. Does not allow ”this” to escape local variables.

```
@SCJAllowed
@SCJMaySelfSuspend(false)
@SCJMayAllocate({})
@SCJPhase({
   javax.safetycritical.annotate.Phase.STARTUP,
   javax.safetycritical.annotate.Phase.INITIALIZATION,
   javax.safetycritical.annotate.Phase.RUN,
   javax.safetycritical.annotate.Phase.CLEANUP })
public boolean isAssignableFrom(Class c)
```

Allocates no memory. Does not allow ”this” or argument ”c” to escape local variables.

```
@SCJAllowed
@SCJMaySelfSuspend(false)
@SCJMayAllocate({})
@SCJPhase({
   javax.safetycritical.annotate.Phase.STARTUP,
   javax.safetycritical.annotate.Phase.INITIALIZATION,
   javax.safetycritical.annotate.Phase.RUN,
   javax.safetycritical.annotate.Phase.CLEANUP })
public boolean isEnum( )
```

Allocates no memory. Does not allow ”this” to escape local variables.

```
@SCJAllowed
@SCJMaySelfSuspend(false)
@SCJMayAllocate({})
@SCJPhase({
   javax.safetycritical.annotate.Phase.STARTUP,
   javax.safetycritical.annotate.Phase.INITIALIZATION,
   javax.safetycritical.annotate.Phase.RUN,
   javax.safetycritical.annotate.Phase.CLEANUP })
public boolean isInstance(Object o)
```

Allocates no memory. Does not allow ”this” to escape local variables.

```
@SCJAllowed
@SCJMaySelfSuspend(false)
@SCJMayAllocate({})
@SCJPhase({
   javax.safetycritical.annotate.Phase.STARTUP,
   javax.safetycritical.annotate.Phase.INITIALIZATION,
   javax.safetycritical.annotate.Phase.RUN,
   javax.safetycritical.annotate.Phase.CLEANUP })
public boolean isInterface( )
```

Allocates no memory. Does not allow ”this” to escape local variables.

```
@SCJAllowed
@SCJMaySelfSuspend(false)
@SCJMayAllocate({})
@SCJPhase({
   javax.safetycritical.annotate.Phase.STARTUP,
   javax.safetycritical.annotate.Phase.INITIALIZATION,
   javax.safetycritical.annotate.Phase.RUN,
   javax.safetycritical.annotate.Phase.CLEANUP })
public boolean isPrimitive( )
```

Allocates no memory. Does not allow ”this” to escape local variables.

```
@SCJAllowed
@SCJMaySelfSuspend(false)
@SCJMayAllocate({javax.safetycritical.annotate.AllocationContext.CURRENT})
@SCJPhase({
   javax.safetycritical.annotate.Phase.STARTUP,
   javax.safetycritical.annotate.Phase.INITIALIZATION,
   javax.safetycritical.annotate.Phase.RUN,
   javax.safetycritical.annotate.Phase.CLEANUP })
public java.lang.String toString( )
```

Does not allow ”this” to escape local variables. Allocates a String and associated internal ”structure” (e.g. char[]) in caller’s scope. (Note: this semantics is desired for consistency with overridden implementation of Object.toString()).

B.3.9 CLASS ClassCastException

```
@SCJAllowed
public class ClassCastException implements java.io.Serializable extends
    java.lang.RuntimeException
```

Thrown to indicate that the code has attempted to cast an object to a subclass of which it is not an instance.

Constructors

```
@SCJAllowed
@SCJMayAllocate({
   javax.safetycritical.annotate.AllocationContext.CURRENT,
   javax.safetycritical.annotate.AllocationContext.INNER,
   javax.safetycritical.annotate.AllocationContext.OUTER})
@SCJMaySelfSuspend(true)
@SCJPhase({
   javax.safetycritical.annotate.Phase.STARTUP,
   javax.safetycritical.annotate.Phase.INITIALIZATION,
   javax.safetycritical.annotate.Phase.RUN,
   javax.safetycritical.annotate.Phase.CLEANUP })
public ClassCastException( )
```

Shall not copy ”this” to any instance or static field.

Allocates an application- and implementation-dependent amount of memory in the current scope (to represent stack backtrace).

```
@SCJAllowed
@SCJMaySelfSuspend(false)
@SCJMayAllocate({javax.safetycritical.annotate.AllocationContext.CURRENT})
@SCJPhase({
   javax.safetycritical.annotate.Phase.STARTUP,
   javax.safetycritical.annotate.Phase.INITIALIZATION,
   javax.safetycritical.annotate.Phase.RUN,
   javax.safetycritical.annotate.Phase.CLEANUP })
public ClassCastException(String msg)
```

Shall not copy ”this” to any instance or static field. The scope containing the msg argument must enclose the scope containing ”this”. Otherwise, an IllegalAssignmentError will be thrown.

Allocates an application- and implementation-dependent amount of memory in the current scope (to represent stack backtrace).

B.3.10 CLASS ClassNotFoundException

```
@SCJAllowed
public class ClassNotFoundException implements java.io.Serializable extends
    java.lang.Exception
```

Thrown when an application tries to load in a class through its string name using the forName method in class Class, the findSystemClass method in class ClassLoader, or the loadClass method in class ClassLoader, but no definition for the class with the specified name could be found.

Constructors

```
@SCJAllowed
@SCJMaySelfSuspend(false)
@SCJMayAllocate({javax.safetycritical.annotate.AllocationContext.CURRENT})
@SCJPhase({
   javax.safetycritical.annotate.Phase.STARTUP,
   javax.safetycritical.annotate.Phase.INITIALIZATION,
   javax.safetycritical.annotate.Phase.RUN,
   javax.safetycritical.annotate.Phase.CLEANUP })
public ClassNotFoundException( )
```

Shall not copy ”this” to any instance or static field.

Allocates an application- and implementation-dependent amount of memory in the current scope (to represent stack backtrace).

```
@SCJAllowed
@SCJMaySelfSuspend(false)
@SCJMayAllocate({javax.safetycritical.annotate.AllocationContext.CURRENT})
@SCJPhase({
   javax.safetycritical.annotate.Phase.STARTUP,
   javax.safetycritical.annotate.Phase.INITIALIZATION,
   javax.safetycritical.annotate.Phase.RUN,
   javax.safetycritical.annotate.Phase.CLEANUP })
public ClassNotFoundException(String msg)
```

Shall not copy ”this” to any instance or static field. The scope containing the msg argument must enclose the scope containing ”this”. Otherwise, an IllegalAssignmentError will be thrown.

Allocates an application- and implementation-dependent amount of memory in the current scope (to represent stack backtrace).

B.3.11 CLASS CloneNotSupportedException

```
@SCJAllowed
public class CloneNotSupportedException implements java.io.Serializable
    extends java.lang.Exception
```

Thrown to indicate that the clone method in class Object has been called to clone an object, but that the object's class does not implement the Cloneable interface.

Constructors

```
@SCJAllowed
@SCJMaySelfSuspend(false)
@SCJMayAllocate({javax.safetycritical.annotate.AllocationContext.CURRENT})
@SCJPhase({
   javax.safetycritical.annotate.Phase.STARTUP,
   javax.safetycritical.annotate.Phase.INITIALIZATION,
   javax.safetycritical.annotate.Phase.RUN,
   javax.safetycritical.annotate.Phase.CLEANUP })
public CloneNotSupportedException( )
```

Shall not copy ”this” to any instance or static field.

Allocates an application- and implementation-dependent amount of memory in the current scope (to represent stack backtrace).

```
@SCJAllowed
@SCJMaySelfSuspend(false)
@SCJMayAllocate({javax.safetycritical.annotate.AllocationContext.CURRENT})
@SCJPhase({
   javax.safetycritical.annotate.Phase.STARTUP,
   javax.safetycritical.annotate.Phase.INITIALIZATION,
   javax.safetycritical.annotate.Phase.RUN,
   javax.safetycritical.annotate.Phase.CLEANUP })
public CloneNotSupportedException(String msg)
```

Shall not copy ”this” to any instance or static field. The scope containing the msg argument must enclose the scope containing ”this”. Otherwise, an IllegalAssignmentError will be thrown.

Allocates an application- and implementation-dependent amount of memory in the current scope (to represent stack backtrace).

B.3.12 CLASS Double

```
@SCJAllowed
public class Double implements java.lang.Comparable<T>, java.io.Serializable
    extends java.lang.Number
```

The Double class wraps a value of the primitive type double in an object. An object of type Double contains a single field whose type is double.

Fields

```
@SCJAllowed
public static final double MAX_EXPONENT
```

```
@SCJAllowed
public static final double MAX_VALUE
```

```
@SCJAllowed
public static final double MIN_EXPONENT
```

```
@SCJAllowed
public static final double MIN_NORMAL
```

```
@SCJAllowed
public static final double MIN_VALUE
```

@SCJAllowed
public static final double NEGATIVE_INFINITY

@SCJAllowed
public static final double NaN

@SCJAllowed
public static final double POSITIVE_INFINITY

@SCJAllowed
public static final int SIZE

@SCJAllowed
public static final java.lang.**Class**<java.lang.**Double**> TYPE

Constructors

@SCJAllowed
@SCJMaySelfSuspend(false)
@SCJMayAllocate({})
@SCJPhase({
javax.safetycritical.annotate.Phase.STARTUP,
javax.safetycritical.annotate.Phase.INITIALIZATION,
javax.safetycritical.annotate.Phase.RUN,
javax.safetycritical.annotate.Phase.CLEANUP })
public Double(**double** val)

Allocates no memory. Does not allow ”this” to escape local variables.

@SCJAllowed
@SCJMaySelfSuspend(false)
@SCJMayAllocate({})
@SCJPhase({
javax.safetycritical.annotate.Phase.STARTUP,
javax.safetycritical.annotate.Phase.INITIALIZATION,
javax.safetycritical.annotate.Phase.RUN,
javax.safetycritical.annotate.Phase.CLEANUP })
public Double(String str)
throws java.lang.NumberFormatException

Allocates no memory. Does not allow ”this” or ”str” argument to escape local variables.

Throws NumberFormatException

Methods

```
@SCJAllowed
@SCJMaySelfSuspend(false)
@SCJMayAllocate({})
@SCJPhase({
    javax.safetycritical.annotate.Phase.STARTUP,
    javax.safetycritical.annotate.Phase.INITIALIZATION,
    javax.safetycritical.annotate.Phase.RUN,
    javax.safetycritical.annotate.Phase.CLEANUP })
public byte byteValue( )
```

Allocates no memory. Does not allow ”this” to escape local variables.

```
@SCJAllowed
@SCJMaySelfSuspend(false)
@SCJMayAllocate({})
@SCJPhase({
    javax.safetycritical.annotate.Phase.STARTUP,
    javax.safetycritical.annotate.Phase.INITIALIZATION,
    javax.safetycritical.annotate.Phase.RUN,
    javax.safetycritical.annotate.Phase.CLEANUP })
public static int compare(double value1, double value2)
```

Allocates no memory.

```
@SCJAllowed
@SCJMaySelfSuspend(false)
@SCJMayAllocate({})
@SCJPhase({
    javax.safetycritical.annotate.Phase.STARTUP,
    javax.safetycritical.annotate.Phase.INITIALIZATION,
    javax.safetycritical.annotate.Phase.RUN,
    javax.safetycritical.annotate.Phase.CLEANUP })
public int compareTo(Double other)
```

Allocates no memory. Does not allow ”this” or ”other” argument to escape local variables.

```
@SCJAllowed
@SCJMaySelfSuspend(false)
@SCJMayAllocate({})
@SCJPhase({
    javax.safetycritical.annotate.Phase.STARTUP,
    javax.safetycritical.annotate.Phase.INITIALIZATION,
    javax.safetycritical.annotate.Phase.RUN,
    javax.safetycritical.annotate.Phase.CLEANUP })
public static long doubleToLongBits(double v)
```

Allocates no memory.

```
@SCJAllowed
@SCJMaySelfSuspend(false)
@SCJMayAllocate({})
@SCJPhase({
    javax.safetycritical.annotate.Phase.STARTUP,
    javax.safetycritical.annotate.Phase.INITIALIZATION,
    javax.safetycritical.annotate.Phase.RUN,
    javax.safetycritical.annotate.Phase.CLEANUP })
public static native long doubleToRawLongBits(double val)
```

Allocates no memory.

```
@SCJAllowed
@SCJMaySelfSuspend(false)
@SCJMayAllocate({})
@SCJPhase({
    javax.safetycritical.annotate.Phase.STARTUP,
    javax.safetycritical.annotate.Phase.INITIALIZATION,
    javax.safetycritical.annotate.Phase.RUN,
    javax.safetycritical.annotate.Phase.CLEANUP })
public double doubleValue( )
```

Allocates no memory. Does not allow ”this” to escape local variables.

```
@SCJAllowed
@SCJMaySelfSuspend(false)
@SCJMayAllocate({})
@SCJPhase({
    javax.safetycritical.annotate.Phase.STARTUP,
    javax.safetycritical.annotate.Phase.INITIALIZATION,
    javax.safetycritical.annotate.Phase.RUN,
    javax.safetycritical.annotate.Phase.CLEANUP })
public boolean equals(Object obj)
```

Allocates no memory. Does not allow ”this” or ”obj” argument to escape local variables.

```
@SCJAllowed
@SCJMaySelfSuspend(false)
@SCJMayAllocate({})
@SCJPhase({
    javax.safetycritical.annotate.Phase.STARTUP,
    javax.safetycritical.annotate.Phase.INITIALIZATION,
    javax.safetycritical.annotate.Phase.RUN,
    javax.safetycritical.annotate.Phase.CLEANUP })
public float floatValue( )
```

Allocates no memory. Does not allow ”this” to escape local variables.

```
@SCJAllowed
@SCJMaySelfSuspend(false)
@SCJMayAllocate({})
@SCJPhase({
    javax.safetycritical.annotate.Phase.STARTUP,
    javax.safetycritical.annotate.Phase.INITIALIZATION,
    javax.safetycritical.annotate.Phase.RUN,
    javax.safetycritical.annotate.Phase.CLEANUP })
public int hashCode( )
```

Allocates no memory. Does not allow ”this” to escape local variables.

```
@SCJAllowed
@SCJMaySelfSuspend(false)
@SCJMayAllocate({})
@SCJPhase({
    javax.safetycritical.annotate.Phase.STARTUP,
    javax.safetycritical.annotate.Phase.INITIALIZATION,
    javax.safetycritical.annotate.Phase.RUN,
    javax.safetycritical.annotate.Phase.CLEANUP })
public int intValue( )
```

Allocates no memory. Does not allow ”this” to escape local variables.

```
@SCJAllowed
@SCJMaySelfSuspend(false)
@SCJMayAllocate({})
@SCJPhase({
    javax.safetycritical.annotate.Phase.STARTUP,
    javax.safetycritical.annotate.Phase.INITIALIZATION,
    javax.safetycritical.annotate.Phase.RUN,
    javax.safetycritical.annotate.Phase.CLEANUP })
public static boolean isInfinite(double v)
```

Allocates no memory.

```
@SCJAllowed
@SCJMaySelfSuspend(false)
@SCJMayAllocate({})
@SCJPhase({
    javax.safetycritical.annotate.Phase.STARTUP,
    javax.safetycritical.annotate.Phase.INITIALIZATION,
    javax.safetycritical.annotate.Phase.RUN,
    javax.safetycritical.annotate.Phase.CLEANUP })
public boolean isInfinite( )
```

Allocates no memory. Does not allow ”this” to escape local variables.

```
@SCJAllowed
@SCJMaySelfSuspend(false)
@SCJMayAllocate({})
@SCJPhase({
    javax.safetycritical.annotate.Phase.STARTUP,
    javax.safetycritical.annotate.Phase.INITIALIZATION,
    javax.safetycritical.annotate.Phase.RUN,
    javax.safetycritical.annotate.Phase.CLEANUP })
public static boolean isNaN(double v)
```

Allocates no memory.

```
@SCJAllowed
@SCJMaySelfSuspend(false)
@SCJMayAllocate({})
@SCJPhase({
    javax.safetycritical.annotate.Phase.STARTUP,
    javax.safetycritical.annotate.Phase.INITIALIZATION,
    javax.safetycritical.annotate.Phase.RUN,
    javax.safetycritical.annotate.Phase.CLEANUP })
public boolean isNaN( )
```

Allocates no memory. Does not allow ”this” to escape local variables.

```
@SCJAllowed
@SCJMaySelfSuspend(false)
@SCJMayAllocate({})
@SCJPhase({
    javax.safetycritical.annotate.Phase.STARTUP,
    javax.safetycritical.annotate.Phase.INITIALIZATION,
    javax.safetycritical.annotate.Phase.RUN,
    javax.safetycritical.annotate.Phase.CLEANUP })
public static double longBitsToDouble(long v)
```

Allocates no memory.

```
@SCJAllowed
@SCJMaySelfSuspend(false)
@SCJMayAllocate({})
@SCJPhase({
    javax.safetycritical.annotate.Phase.STARTUP,
    javax.safetycritical.annotate.Phase.INITIALIZATION,
    javax.safetycritical.annotate.Phase.RUN,
    javax.safetycritical.annotate.Phase.CLEANUP })
public long longValue( )
```

Allocates no memory. Does not allow ”this” to escape local variables.

```
@SCJAllowed
@SCJMaySelfSuspend(false)
@SCJMayAllocate({})
@SCJPhase({
    javax.safetycritical.annotate.Phase.STARTUP,
    javax.safetycritical.annotate.Phase.INITIALIZATION,
    javax.safetycritical.annotate.Phase.RUN,
    javax.safetycritical.annotate.Phase.CLEANUP })
public static double parseDouble(String s)
  throws java.lang.NumberFormatException
```

Does not allow ”this” to escape local variables.

Throws NumberFormatException

```
@SCJAllowed
@SCJMaySelfSuspend(false)
@SCJMayAllocate({})
@SCJPhase({
    javax.safetycritical.annotate.Phase.STARTUP,
    javax.safetycritical.annotate.Phase.INITIALIZATION,
    javax.safetycritical.annotate.Phase.RUN,
    javax.safetycritical.annotate.Phase.CLEANUP })
public short shortValue( )
```

Allocates no memory. Does not allow ”this” to escape local variables.

```
@SCJAllowed
@SCJMaySelfSuspend(false)
@SCJMayAllocate({javax.safetycritical.annotate.AllocationContext.CURRENT})
@SCJPhase({
    javax.safetycritical.annotate.Phase.STARTUP,
    javax.safetycritical.annotate.Phase.INITIALIZATION,
    javax.safetycritical.annotate.Phase.RUN,
    javax.safetycritical.annotate.Phase.CLEANUP })
public static java.lang.String toString(double v)
```

Allocates a String and associated internal ”structure” (e.g. char[]) in caller’s scope.

```
@SCJAllowed
@SCJMaySelfSuspend(false)
@SCJMayAllocate({javax.safetycritical.annotate.AllocationContext.CURRENT})
```

```
@SCJPhase({
    javax.safetycritical.annotate.Phase.STARTUP,
    javax.safetycritical.annotate.Phase.INITIALIZATION,
    javax.safetycritical.annotate.Phase.RUN,
    javax.safetycritical.annotate.Phase.CLEANUP })
public java.lang.String toString( )
```

Does not allow ”this” to escape local variables. Allocates a String and associated internal ”structure” (e.g. char[]) in caller’s scope. (Note: this semantics is desired for consistency with overridden implementation of Object.toString()).

```
@SCJAllowed
@SCJMaySelfSuspend(false)
@SCJMayAllocate({javax.safetycritical.annotate.AllocationContext.CURRENT})
@SCJPhase({
    javax.safetycritical.annotate.Phase.STARTUP,
    javax.safetycritical.annotate.Phase.INITIALIZATION,
    javax.safetycritical.annotate.Phase.RUN,
    javax.safetycritical.annotate.Phase.CLEANUP })
public static java.lang.Double valueOf(String str)
  throws java.lang.NumberFormatException
```

Does not allow ”this” to escape local variables. Allocates a Double in caller’s scope.

Throws NumberFormatException

```
@SCJAllowed
@SCJMaySelfSuspend(false)
@SCJMayAllocate({javax.safetycritical.annotate.AllocationContext.CURRENT})
@SCJPhase({
    javax.safetycritical.annotate.Phase.STARTUP,
    javax.safetycritical.annotate.Phase.INITIALIZATION,
    javax.safetycritical.annotate.Phase.RUN,
    javax.safetycritical.annotate.Phase.CLEANUP })
public static java.lang.Double valueOf(double val)
```

Allocates a Double in caller’s scope.

B.3.13 CLASS Enum

```
@SCJAllowed
public abstract class Enum<E extends Enum<E>> implements
    java.lang.Comparable<T>, java.io.Serializable extends java.lang.Object
```

This is the common base class of all Java language enumeration types.

Constructors

```
@SCJAllowed
@SCJMaySelfSuspend(false)
@SCJMayAllocate({})
@SCJPhase({
    javax.safetycritical.annotate.Phase.STARTUP,
    javax.safetycritical.annotate.Phase.INITIALIZATION,
    javax.safetycritical.annotate.Phase.RUN,
    javax.safetycritical.annotate.Phase.CLEANUP })
protected Enum(String name, int ordinal)
```

Allocates no memory. Does not allow ”this” to escape local variables. Requires that ”name” argument reside in a scope that enclosees the scope of ”this”.

name — The name of this enum constant, which is the identifier used to declare it.

ordinal — The ordinal of this enumeration constant (its position in the enum declaration, where the initial constant is assigned an ordinal of zero).

Methods

```
@SCJAllowed
@SCJMaySelfSuspend(false)
@SCJMayAllocate({})
@SCJPhase({
    javax.safetycritical.annotate.Phase.STARTUP,
    javax.safetycritical.annotate.Phase.INITIALIZATION,
    javax.safetycritical.annotate.Phase.RUN,
    javax.safetycritical.annotate.Phase.CLEANUP })
public final int compareTo(E o)
```

Allocates no memory. Does not allow ”this” or ”o” argument to escape local variables.

o — The object to be compared.

returns a negative integer, zero, or a positive integer as this object is less than, equal to, or greater than the specified object.

```
@SCJAllowed
@SCJMaySelfSuspend(false)
@SCJMayAllocate({})
@SCJPhase({
    javax.safetycritical.annotate.Phase.STARTUP,
    javax.safetycritical.annotate.Phase.INITIALIZATION,
    javax.safetycritical.annotate.Phase.RUN,
    javax.safetycritical.annotate.Phase.CLEANUP })
public final boolean equals(Object o)
```

Allocates no memory. Does not allow ”this” or ”o” argument to escape local variables.

o — The object to be compared for equality with this object.

returns true if the specified object is equal to this enum constant.

```
@SCJAllowed
@SCJMaySelfSuspend(false)
@SCJMayAllocate({})
@SCJPhase({
   javax.safetycritical.annotate.Phase.STARTUP,
   javax.safetycritical.annotate.Phase.INITIALIZATION,
   javax.safetycritical.annotate.Phase.RUN,
   javax.safetycritical.annotate.Phase.CLEANUP })
public final java.lang.Class<E> getDeclaringClass( )
```

Allocates no memory. Returns a reference to a previously allocated Class, which resides in its ClassLoader scope.

returns The Class object corresponding to this enum constant's enum type

```
@SCJAllowed
@SCJMaySelfSuspend(false)
@SCJMayAllocate({})
@SCJPhase({
   javax.safetycritical.annotate.Phase.STARTUP,
   javax.safetycritical.annotate.Phase.INITIALIZATION,
   javax.safetycritical.annotate.Phase.RUN,
   javax.safetycritical.annotate.Phase.CLEANUP })
public final int hashCode( )
```

Allocates no memory. Does not allow ”this” to escape local variables.

returns A hash code for this enum constant.

```
@SCJAllowed
@SCJMaySelfSuspend(false)
@SCJMayAllocate({})
@SCJPhase({
   javax.safetycritical.annotate.Phase.STARTUP,
   javax.safetycritical.annotate.Phase.INITIALIZATION,
   javax.safetycritical.annotate.Phase.RUN,
   javax.safetycritical.annotate.Phase.CLEANUP })
public final java.lang.String name( )
```

Allocates no memory. Returns a reference to this enumeration constant's previously allocated String name. The String resides in the corresponding ClassLoader scope.

returns the name of this enum constant

```
@SCJAllowed
@SCJMaySelfSuspend(false)
@SCJMayAllocate({})
@SCJPhase({
    javax.safetycritical.annotate.Phase.STARTUP,
    javax.safetycritical.annotate.Phase.INITIALIZATION,
    javax.safetycritical.annotate.Phase.RUN,
    javax.safetycritical.annotate.Phase.CLEANUP })
public final int ordinal( )
```

Allocates no memory. Does not allow ”this” to escape local variables.

returns the ordinal of this enumeration constant

```
@SCJAllowed
@SCJMaySelfSuspend(false)
@SCJMayAllocate({javax.safetycritical.annotate.AllocationContext.CURRENT})
@SCJPhase({
    javax.safetycritical.annotate.Phase.STARTUP,
    javax.safetycritical.annotate.Phase.INITIALIZATION,
    javax.safetycritical.annotate.Phase.RUN,
    javax.safetycritical.annotate.Phase.CLEANUP })
public java.lang.String toString( )
```

Does not allow ”this” to escape local variables. Allocates a String and associated internal ”structure” (e.g. char[]) in caller’s scope. (Note: this semantics is desired for consistency with overridden implementation of Object.toString()).

returns The name of this enum constant, as contained in the declaration.

B.3.14 CLASS Error

```
@SCJAllowed
public class Error extends java.lang.Throwable
```

An Error is a subclass of Throwable that indicates serious problems that a reasonable application should not try to catch.

Constructors

```
@SCJAllowed
@SCJMayAllocate({javax.safetycritical.annotate.AllocationContext.CURRENT})
@SCJMaySelfSuspend(false)
```

```
@SCJPhase({
   javax.safetycritical.annotate.Phase.STARTUP,
   javax.safetycritical.annotate.Phase.INITIALIZATION,
   javax.safetycritical.annotate.Phase.RUN,
   javax.safetycritical.annotate.Phase.CLEANUP })
public Error(String msg, Throwable cause)
```

Constructs an Error object with a specified detail message and with a specified cause. If cause is null, Services.captureStackBacktrace(this) is called to save the backtrace associated with the current thread. If cause is not null, Services.captureStackBacktrace(this) is not called to avoid overwriting the backtrace associated with the cause.

Allocates an application- and implementation-defined amount of memory in the current scope (to represent stack backtrace).

msg — the detail message for this Error object.

cause — the exception that caused this error.

```
@SCJAllowed
@SCJMayAllocate({javax.safetycritical.annotate.AllocationContext.CURRENT})
@SCJMaySelfSuspend(false)
@SCJPhase({
   javax.safetycritical.annotate.Phase.STARTUP,
   javax.safetycritical.annotate.Phase.INITIALIZATION,
   javax.safetycritical.annotate.Phase.RUN,
   javax.safetycritical.annotate.Phase.CLEANUP })
public Error( )
```

This constructor behaves the same as calling Error(String, Throwable) with the arguments (null, null).

```
@SCJAllowed
@SCJMayAllocate({javax.safetycritical.annotate.AllocationContext.CURRENT})
@SCJMaySelfSuspend(false)
@SCJPhase({
   javax.safetycritical.annotate.Phase.STARTUP,
   javax.safetycritical.annotate.Phase.INITIALIZATION,
   javax.safetycritical.annotate.Phase.RUN,
   javax.safetycritical.annotate.Phase.CLEANUP })
public Error(String msg)
```

This constructor behaves the same as calling Error(String, Throwable) with the arguments (msg, null).

```
@SCJAllowed
@SCJMayAllocate({javax.safetycritical.annotate.AllocationContext.CURRENT})
@SCJMaySelfSuspend(false)
@SCJPhase({
   javax.safetycritical.annotate.Phase.STARTUP,
   javax.safetycritical.annotate.Phase.INITIALIZATION,
   javax.safetycritical.annotate.Phase.RUN,
   javax.safetycritical.annotate.Phase.CLEANUP })
public Error(Throwable cause)
```

This constructor behaves the same as calling Error(String, Throwable) with the arguments (null, cause).

B.3.15 CLASS Exception

```
@SCJAllowed
public class Exception implements java.io.Serializable extends
    java.lang.Throwable
```

The class Exception and its subclasses are a form of Throwable that indicates conditions that a reasonable application might want to catch.

Constructors

```
@SCJAllowed
@SCJMaySelfSuspend(false)
@SCJMayAllocate({})
@SCJPhase({
   javax.safetycritical.annotate.Phase.STARTUP,
   javax.safetycritical.annotate.Phase.INITIALIZATION,
   javax.safetycritical.annotate.Phase.RUN,
   javax.safetycritical.annotate.Phase.CLEANUP })
public Exception(String msg, Throwable cause)
```

Constructs an Exception object with an optional detail message and an optional cause. If cause is null, Services.captureStackBacktrace(this) is called to save the backtrace associated with the current thread. If cause is not null, Services.captureStackBacktrace(this) is not called to avoid overwriting the backtrace associated with the cause.

msg — the detail message for this Exception object.

cause — the cause of this exception .

```
@SCJAllowed
@SCJMaySelfSuspend(false)
@SCJMayAllocate({})
@SCJPhase({
    javax.safetycritical.annotate.Phase.STARTUP,
    javax.safetycritical.annotate.Phase.INITIALIZATION,
    javax.safetycritical.annotate.Phase.RUN,
    javax.safetycritical.annotate.Phase.CLEANUP })
public Exception( )
```

This constructor behaves the same as calling Exception(null, null).

```
@SCJAllowed
@SCJMaySelfSuspend(false)
@SCJMayAllocate({})
@SCJPhase({
    javax.safetycritical.annotate.Phase.STARTUP,
    javax.safetycritical.annotate.Phase.INITIALIZATION,
    javax.safetycritical.annotate.Phase.RUN,
    javax.safetycritical.annotate.Phase.CLEANUP })
public Exception(String msg)
```

This constructor behaves the same as calling Exception(msg, null).

```
@SCJAllowed
@SCJMaySelfSuspend(false)
@SCJMayAllocate({})
@SCJPhase({
    javax.safetycritical.annotate.Phase.STARTUP,
    javax.safetycritical.annotate.Phase.INITIALIZATION,
    javax.safetycritical.annotate.Phase.RUN,
    javax.safetycritical.annotate.Phase.CLEANUP })
public Exception(Throwable cause)
```

This constructor behaves the same as calling Exception(null, cause).

B.3.16 CLASS ExceptionInInitializerError

```
@SCJAllowed
public class ExceptionInInitializerError extends java.lang.Exception
```

Signals that an unexpected exception has occurred in a static initializer. An ExceptionInInitializerError is thrown to indicate that an exception occurred during evaluation of a static initializer or the initializer for a static variable.

Constructors

```
@SCJAllowed
@SCJMayAllocate({
    javax.safetycritical.annotate.AllocationContext.CURRENT,
    javax.safetycritical.annotate.AllocationContext.INNER,
    javax.safetycritical.annotate.AllocationContext.OUTER})
@SCJMaySelfSuspend(true)
@SCJPhase({
    javax.safetycritical.annotate.Phase.STARTUP,
    javax.safetycritical.annotate.Phase.INITIALIZATION,
    javax.safetycritical.annotate.Phase.RUN,
    javax.safetycritical.annotate.Phase.CLEANUP })
public ExceptionInInitializerError( )
```

```
@SCJAllowed
@SCJMayAllocate({
    javax.safetycritical.annotate.AllocationContext.CURRENT,
    javax.safetycritical.annotate.AllocationContext.INNER,
    javax.safetycritical.annotate.AllocationContext.OUTER})
@SCJMaySelfSuspend(true)
@SCJPhase({
    javax.safetycritical.annotate.Phase.STARTUP,
    javax.safetycritical.annotate.Phase.INITIALIZATION,
    javax.safetycritical.annotate.Phase.RUN,
    javax.safetycritical.annotate.Phase.CLEANUP })
public ExceptionInInitializerError(String msg)
```

```
@SCJAllowed
@SCJMayAllocate({
    javax.safetycritical.annotate.AllocationContext.CURRENT,
    javax.safetycritical.annotate.AllocationContext.INNER,
    javax.safetycritical.annotate.AllocationContext.OUTER})
@SCJMaySelfSuspend(true)
@SCJPhase({
    javax.safetycritical.annotate.Phase.STARTUP,
    javax.safetycritical.annotate.Phase.INITIALIZATION,
    javax.safetycritical.annotate.Phase.RUN,
    javax.safetycritical.annotate.Phase.CLEANUP })
public ExceptionInInitializerError(Throwable cause)
```

```
@SCJAllowed
@SCJMayAllocate({
    javax.safetycritical.annotate.AllocationContext.CURRENT,
    javax.safetycritical.annotate.AllocationContext.INNER,
    javax.safetycritical.annotate.AllocationContext.OUTER})
@SCJMaySelfSuspend(true)
```

```
@SCJPhase({
    javax.safetycritical.annotate.Phase.STARTUP,
    javax.safetycritical.annotate.Phase.INITIALIZATION,
    javax.safetycritical.annotate.Phase.RUN,
    javax.safetycritical.annotate.Phase.CLEANUP })
public ExceptionInInitializerError(String msg, Throwable cause)
```

B.3.17 CLASS Float

```
@SCJAllowed
public class Float implements java.lang.Comparable<T>, java.io.Serializable
    extends java.lang.Number
```

The Float class wraps a value of primitive type float in an object. An object of type Float contains a single field whose type is float.

Fields

```
@SCJAllowed
public static final float MAX_EXPONENT
```

```
@SCJAllowed
public static final float MAX_VALUE
```

```
@SCJAllowed
public static final float MIN_EXPONENT
```

```
@SCJAllowed
public static final float MIN_NORMAL
```

```
@SCJAllowed
public static final float MIN_VALUE
```

```
@SCJAllowed
public static final float NEGATIVE_INFINITY
```

```
@SCJAllowed
public static final float NaN
```

```
@SCJAllowed
public static final float POSITIVE_INFINITY
```

```
@SCJAllowed
public static final int SIZE
```

```
@SCJAllowed
public static final java.lang.Class<java.lang.Float> TYPE
```

Constructors

```
@SCJAllowed
@SCJMaySelfSuspend(false)
@SCJMayAllocate({})
@SCJPhase({
    javax.safetycritical.annotate.Phase.STARTUP,
    javax.safetycritical.annotate.Phase.INITIALIZATION,
    javax.safetycritical.annotate.Phase.RUN,
    javax.safetycritical.annotate.Phase.CLEANUP })
public Float(float val)
```

Allocates no memory. Does not allow ”this” to escape local variables.

```
@SCJAllowed
@SCJMaySelfSuspend(false)
@SCJMayAllocate({})
@SCJPhase({
    javax.safetycritical.annotate.Phase.STARTUP,
    javax.safetycritical.annotate.Phase.INITIALIZATION,
    javax.safetycritical.annotate.Phase.RUN,
    javax.safetycritical.annotate.Phase.CLEANUP })
public Float(double val)
```

Allocates no memory. Does not allow ”this” to escape local variables.

```
@SCJAllowed
@SCJMaySelfSuspend(false)
@SCJMayAllocate({})
@SCJPhase({
    javax.safetycritical.annotate.Phase.STARTUP,
    javax.safetycritical.annotate.Phase.INITIALIZATION,
    javax.safetycritical.annotate.Phase.RUN,
    javax.safetycritical.annotate.Phase.CLEANUP })
public Float(String str)
  throws java.lang.NumberFormatException
```

Allocates no memory. Does not allow ”this” or ”str” argument to escape local variables.

Throws NumberFormatException

Methods

```
@SCJAllowed
@SCJMaySelfSuspend(false)
@SCJMayAllocate({})
@SCJPhase({
    javax.safetycritical.annotate.Phase.STARTUP,
    javax.safetycritical.annotate.Phase.INITIALIZATION,
    javax.safetycritical.annotate.Phase.RUN,
    javax.safetycritical.annotate.Phase.CLEANUP })
public byte byteValue( )
```

Allocates no memory. Does not allow ”this” to escape local variables.

```
@SCJAllowed
@SCJMaySelfSuspend(false)
@SCJMayAllocate({})
@SCJPhase({
    javax.safetycritical.annotate.Phase.STARTUP,
    javax.safetycritical.annotate.Phase.INITIALIZATION,
    javax.safetycritical.annotate.Phase.RUN,
    javax.safetycritical.annotate.Phase.CLEANUP })
public static int compare(float value1, float value2)
```

Allocates no memory.

```
@SCJAllowed
@SCJMaySelfSuspend(false)
@SCJMayAllocate({})
@SCJPhase({
    javax.safetycritical.annotate.Phase.STARTUP,
    javax.safetycritical.annotate.Phase.INITIALIZATION,
    javax.safetycritical.annotate.Phase.RUN,
    javax.safetycritical.annotate.Phase.CLEANUP })
public int compareTo(Float other)
```

Allocates no memory. Does not allow ”this” or ”other” argument to escape local variables.

```
@SCJAllowed
@SCJMaySelfSuspend(false)
@SCJMayAllocate({})
@SCJPhase({
```

```
    javax.safetycritical.annotate.Phase.STARTUP,
    javax.safetycritical.annotate.Phase.INITIALIZATION,
    javax.safetycritical.annotate.Phase.RUN,
    javax.safetycritical.annotate.Phase.CLEANUP })
public double doubleValue( )
```

Allocates no memory. Does not allow ”this” to escape local variables.

```
@SCJAllowed
@SCJMaySelfSuspend(false)
@SCJMayAllocate({})
@SCJPhase({
    javax.safetycritical.annotate.Phase.STARTUP,
    javax.safetycritical.annotate.Phase.INITIALIZATION,
    javax.safetycritical.annotate.Phase.RUN,
    javax.safetycritical.annotate.Phase.CLEANUP })
public boolean equals(Object obj)
```

Allocates no memory. Does not allow ”this” or ”obj” argument to escape local variables.

```
@SCJAllowed
@SCJMaySelfSuspend(false)
@SCJMayAllocate({})
@SCJPhase({
    javax.safetycritical.annotate.Phase.STARTUP,
    javax.safetycritical.annotate.Phase.INITIALIZATION,
    javax.safetycritical.annotate.Phase.RUN,
    javax.safetycritical.annotate.Phase.CLEANUP })
public static int floatToIntBits(float v)
```

Allocates no memory.

```
@SCJAllowed
@SCJMaySelfSuspend(false)
@SCJMayAllocate({})
@SCJPhase({
    javax.safetycritical.annotate.Phase.STARTUP,
    javax.safetycritical.annotate.Phase.INITIALIZATION,
    javax.safetycritical.annotate.Phase.RUN,
    javax.safetycritical.annotate.Phase.CLEANUP })
public static int floatToRawIntBits(float v)
```

Allocates no memory.

```
@SCJAllowed
@SCJMaySelfSuspend(false)
@SCJMayAllocate({})
@SCJPhase({
    javax.safetycritical.annotate.Phase.STARTUP,
    javax.safetycritical.annotate.Phase.INITIALIZATION,
    javax.safetycritical.annotate.Phase.RUN,
    javax.safetycritical.annotate.Phase.CLEANUP })
public float floatValue( )
```

Allocates no memory. Does not allow ”this” to escape local variables.

```
@SCJAllowed
@SCJMaySelfSuspend(false)
@SCJMayAllocate({})
@SCJPhase({
    javax.safetycritical.annotate.Phase.STARTUP,
    javax.safetycritical.annotate.Phase.INITIALIZATION,
    javax.safetycritical.annotate.Phase.RUN,
    javax.safetycritical.annotate.Phase.CLEANUP })
public int hashCode( )
```

Allocates no memory. Does not allow ”this” to escape local variables.

```
@SCJAllowed
@SCJMaySelfSuspend(false)
@SCJMayAllocate({})
@SCJPhase({
    javax.safetycritical.annotate.Phase.STARTUP,
    javax.safetycritical.annotate.Phase.INITIALIZATION,
    javax.safetycritical.annotate.Phase.RUN,
    javax.safetycritical.annotate.Phase.CLEANUP })
public static float intBitsToFloat(int v)
```

Allocates no memory.

```
@SCJAllowed
@SCJMaySelfSuspend(false)
@SCJMayAllocate({})
@SCJPhase({
    javax.safetycritical.annotate.Phase.STARTUP,
    javax.safetycritical.annotate.Phase.INITIALIZATION,
    javax.safetycritical.annotate.Phase.RUN,
    javax.safetycritical.annotate.Phase.CLEANUP })
public int intValue( )
```

Allocates no memory. Does not allow ”this” to escape local variables.

```
@SCJAllowed
@SCJMaySelfSuspend(false)
@SCJMayAllocate({})
@SCJPhase({
    javax.safetycritical.annotate.Phase.STARTUP,
    javax.safetycritical.annotate.Phase.INITIALIZATION,
    javax.safetycritical.annotate.Phase.RUN,
    javax.safetycritical.annotate.Phase.CLEANUP })
public static boolean isInfinite(float v)
```

Allocates no memory.

```
@SCJAllowed
@SCJMaySelfSuspend(false)
@SCJMayAllocate({})
@SCJPhase({
    javax.safetycritical.annotate.Phase.STARTUP,
    javax.safetycritical.annotate.Phase.INITIALIZATION,
    javax.safetycritical.annotate.Phase.RUN,
    javax.safetycritical.annotate.Phase.CLEANUP })
public boolean isInfinite( )
```

Allocates no memory. Does not allow ”this” to escape local variables.

```
@SCJAllowed
@SCJMaySelfSuspend(false)
@SCJMayAllocate({})
@SCJPhase({
    javax.safetycritical.annotate.Phase.STARTUP,
    javax.safetycritical.annotate.Phase.INITIALIZATION,
    javax.safetycritical.annotate.Phase.RUN,
    javax.safetycritical.annotate.Phase.CLEANUP })
public static boolean isNaN(float v)
```

Allocates no memory.

```
@SCJAllowed
@SCJMaySelfSuspend(false)
@SCJMayAllocate({})
@SCJPhase({
    javax.safetycritical.annotate.Phase.STARTUP,
    javax.safetycritical.annotate.Phase.INITIALIZATION,
    javax.safetycritical.annotate.Phase.RUN,
    javax.safetycritical.annotate.Phase.CLEANUP })
public boolean isNaN( )
```

Allocates no memory. Does not allow ”this” to escape local variables.

```
@SCJAllowed
@SCJMaySelfSuspend(false)
@SCJMayAllocate({})
@SCJPhase({
    javax.safetycritical.annotate.Phase.STARTUP,
    javax.safetycritical.annotate.Phase.INITIALIZATION,
    javax.safetycritical.annotate.Phase.RUN,
    javax.safetycritical.annotate.Phase.CLEANUP })
public long longValue( )
```

Allocates no memory. Does not allow ”this” to escape local variables.

```
@SCJAllowed
@SCJMaySelfSuspend(false)
@SCJMayAllocate({})
@SCJPhase({
    javax.safetycritical.annotate.Phase.STARTUP,
    javax.safetycritical.annotate.Phase.INITIALIZATION,
    javax.safetycritical.annotate.Phase.RUN,
    javax.safetycritical.annotate.Phase.CLEANUP })
public static float parseFloat(String s)
  throws java.lang.NumberFormatException
```

Allocates no memory. Does not allow ”s” argument to escape local variables.

Throws NumberFormatException

```
@SCJAllowed
@SCJMaySelfSuspend(false)
@SCJMayAllocate({})
@SCJPhase({
    javax.safetycritical.annotate.Phase.STARTUP,
    javax.safetycritical.annotate.Phase.INITIALIZATION,
    javax.safetycritical.annotate.Phase.RUN,
    javax.safetycritical.annotate.Phase.CLEANUP })
public short shortValue( )
```

Allocates no memory. Does not allow ”this” to escape local variables.

```
@SCJAllowed
@SCJMaySelfSuspend(false)
@SCJMayAllocate({javax.safetycritical.annotate.AllocationContext.CURRENT})
@SCJPhase({
```

```
    javax.safetycritical.annotate.Phase.STARTUP,
    javax.safetycritical.annotate.Phase.INITIALIZATION,
    javax.safetycritical.annotate.Phase.RUN,
    javax.safetycritical.annotate.Phase.CLEANUP })
public static java.lang.String toHexString(float v)
```

Allocates a String and associated internal ”structure” (e.g. char[]) in caller’s scope.

```
@SCJAllowed
@SCJMaySelfSuspend(false)
@SCJMayAllocate({javax.safetycritical.annotate.AllocationContext.CURRENT})
@SCJPhase({
    javax.safetycritical.annotate.Phase.STARTUP,
    javax.safetycritical.annotate.Phase.INITIALIZATION,
    javax.safetycritical.annotate.Phase.RUN,
    javax.safetycritical.annotate.Phase.CLEANUP })
public static java.lang.String toString(float v)
```

Allocates a String and associated internal ”structure” (e.g. char[]) in caller’s scope.

```
@SCJAllowed
@SCJMaySelfSuspend(false)
@SCJMayAllocate({javax.safetycritical.annotate.AllocationContext.CURRENT})
@SCJPhase({
    javax.safetycritical.annotate.Phase.STARTUP,
    javax.safetycritical.annotate.Phase.INITIALIZATION,
    javax.safetycritical.annotate.Phase.RUN,
    javax.safetycritical.annotate.Phase.CLEANUP })
public java.lang.String toString( )
```

Does not allow ”this” to escape local variables. Allocates a String and associated internal ”structure” (e.g. char[]) in caller’s scope. (Note: this semantics is desired for consistency with overridden implementation of Object.toString()).

```
@SCJAllowed
@SCJMaySelfSuspend(false)
@SCJMayAllocate({javax.safetycritical.annotate.AllocationContext.CURRENT})
@SCJPhase({
    javax.safetycritical.annotate.Phase.STARTUP,
    javax.safetycritical.annotate.Phase.INITIALIZATION,
    javax.safetycritical.annotate.Phase.RUN,
    javax.safetycritical.annotate.Phase.CLEANUP })
public static java.lang.Float valueOf(String str)
  throws java.lang.NumberFormatException
```

Does not allow ”this” to escape local variables. Allocates a Float in caller’s scope.

Throws NumberFormatException

```
@SCJAllowed
@SCJMaySelfSuspend(false)
@SCJMayAllocate({javax.safetycritical.annotate.AllocationContext.CURRENT})
@SCJPhase({
   javax.safetycritical.annotate.Phase.STARTUP,
   javax.safetycritical.annotate.Phase.INITIALIZATION,
   javax.safetycritical.annotate.Phase.RUN,
   javax.safetycritical.annotate.Phase.CLEANUP })
public static java.lang.Float valueOf(float val)
```

Allocates a Float in caller’s scope.

B.3.18 CLASS IllegalArgumentException

```
@SCJAllowed
public class IllegalArgumentException implements java.io.Serializable extends
    java.lang.RuntimeException
```

Thrown to indicate that a method has been passed an illegal or inappropriate argument.

Constructors

```
@SCJAllowed
@SCJMaySelfSuspend(false)
@SCJMayAllocate({})
@SCJPhase({
   javax.safetycritical.annotate.Phase.STARTUP,
   javax.safetycritical.annotate.Phase.INITIALIZATION,
   javax.safetycritical.annotate.Phase.RUN,
   javax.safetycritical.annotate.Phase.CLEANUP })
public IllegalArgumentException( )
```

Shall not copy ”this” to any instance or static field.

Allocates an application- and implementation-dependent amount of memory in the current scope (to represent stack backtrace).

```
@SCJAllowed
@SCJMaySelfSuspend(false)
@SCJMayAllocate({})
@SCJPhase({
   javax.safetycritical.annotate.Phase.STARTUP,
   javax.safetycritical.annotate.Phase.INITIALIZATION,
   javax.safetycritical.annotate.Phase.RUN,
   javax.safetycritical.annotate.Phase.CLEANUP })
public IllegalArgumentException(String msg)
```

Shall not copy ”this” to any instance or static field. The scope containing the msg argument must enclose the scope containing ”this”. Otherwise, an IllegalAssignmentError will be thrown.

Allocates an application- and implementation-dependent amount of memory in the current scope (to represent stack backtrace).

```
@SCJAllowed
@SCJMaySelfSuspend(false)
@SCJMayAllocate({})
@SCJPhase({
   javax.safetycritical.annotate.Phase.STARTUP,
   javax.safetycritical.annotate.Phase.INITIALIZATION,
   javax.safetycritical.annotate.Phase.RUN,
   javax.safetycritical.annotate.Phase.CLEANUP })
public IllegalArgumentException(String msg, Throwable t)
```

Shall not copy ”this” to any instance or static field. The scope containing the msg argument must enclose the scope containing ”this”. Otherwise, an IllegalAssignmentError will be thrown.

Allocates an application- and implementation-dependent amount of memory in the current scope (to represent stack backtrace).

```
@SCJAllowed
@SCJMaySelfSuspend(false)
@SCJMayAllocate({})
@SCJPhase({
   javax.safetycritical.annotate.Phase.STARTUP,
   javax.safetycritical.annotate.Phase.INITIALIZATION,
   javax.safetycritical.annotate.Phase.RUN,
   javax.safetycritical.annotate.Phase.CLEANUP })
public IllegalArgumentException(Throwable t)
```

Shall not copy ”this” to any instance or static field. The scope containing the msg argument must enclose the scope containing ”this”. Otherwise, an IllegalAssignmentError will be thrown.

Allocates an application- and implementation-dependent amount of memory in the current scope (to represent stack backtrace).

B.3.19 CLASS IllegalMonitorStateException

```
@SCJAllowed(javax.safetycritical.annotate.Level.LEVEL_2)
public class IllegalMonitorStateException implements java.io.Serializable
    extends java.lang.RuntimeException
```

Thrown to indicate that a thread has attempted to wait on an object's monitor or to notify other threads waiting on an object's monitor without owning the specified monitor.

Constructors

```
@SCJAllowed
@SCJMaySelfSuspend(false)
@SCJMayAllocate({})
@SCJPhase({
   javax.safetycritical.annotate.Phase.STARTUP,
   javax.safetycritical.annotate.Phase.INITIALIZATION,
   javax.safetycritical.annotate.Phase.RUN,
   javax.safetycritical.annotate.Phase.CLEANUP })
public IllegalMonitorStateException( )
```

Shall not copy "this" to any instance or static field.

Allocates an application- and implementation-dependent amount of memory in the current scope (to represent stack backtrace).

```
@SCJAllowed
@SCJMaySelfSuspend(false)
@SCJMayAllocate({})
@SCJPhase({
   javax.safetycritical.annotate.Phase.STARTUP,
   javax.safetycritical.annotate.Phase.INITIALIZATION,
   javax.safetycritical.annotate.Phase.RUN,
   javax.safetycritical.annotate.Phase.CLEANUP })
public IllegalMonitorStateException(String msg)
```

Shall not copy "this" to any instance or static field. The scope containing the msg argument must enclose the scope containing "this". Otherwise, an IllegalAssignmentError will be thrown.

Allocates an application- and implementation-dependent amount of memory in the current scope (to represent stack backtrace).

```
@SCJAllowed
@SCJMaySelfSuspend(false)
@SCJMayAllocate({})
@SCJPhase({
   javax.safetycritical.annotate.Phase.STARTUP,
   javax.safetycritical.annotate.Phase.INITIALIZATION,
   javax.safetycritical.annotate.Phase.RUN,
   javax.safetycritical.annotate.Phase.CLEANUP })
public IllegalMonitorStateException(String msg, Throwable t)
```

Shall not copy ”this” to any instance or static field. The scope containing the msg argument must enclose the scope containing ”this”. Otherwise, an IllegalAssignmentError will be thrown.

Allocates an application- and implementation-dependent amount of memory in the current scope (to represent stack backtrace).

```
@SCJAllowed
@SCJMaySelfSuspend(false)
@SCJMayAllocate({})
@SCJPhase({
   javax.safetycritical.annotate.Phase.STARTUP,
   javax.safetycritical.annotate.Phase.INITIALIZATION,
   javax.safetycritical.annotate.Phase.RUN,
   javax.safetycritical.annotate.Phase.CLEANUP })
public IllegalMonitorStateException(Throwable t)
```

Shall not copy ”this” to any instance or static field. The scope containing the msg argument must enclose the scope containing ”this”. Otherwise, an IllegalAssignmentError will be thrown.

Allocates an application- and implementation-dependent amount of memory in the current scope (to represent stack backtrace).

B.3.20 CLASS IllegalStateException

```
@SCJAllowed
public class IllegalStateException implements java.io.Serializable extends
    java.lang.RuntimeException
```

Signals that a method has been invoked at an illegal or inappropriate time. In other words, the Java environment or Java application is not in an appropriate state for the requested operation.

Constructors

```
@SCJAllowed
@SCJMaySelfSuspend(false)
@SCJMayAllocate({})
@SCJPhase({
   javax.safetycritical.annotate.Phase.STARTUP,
   javax.safetycritical.annotate.Phase.INITIALIZATION,
   javax.safetycritical.annotate.Phase.RUN,
   javax.safetycritical.annotate.Phase.CLEANUP })
public IllegalStateException( )
```

Shall not copy ”this” to any instance or static field.

Allocates an application- and implementation-dependent amount of memory in the current scope (to represent stack backtrace).

```
@SCJAllowed
@SCJMaySelfSuspend(false)
@SCJMayAllocate({})
@SCJPhase({
   javax.safetycritical.annotate.Phase.STARTUP,
   javax.safetycritical.annotate.Phase.INITIALIZATION,
   javax.safetycritical.annotate.Phase.RUN,
   javax.safetycritical.annotate.Phase.CLEANUP })
public IllegalStateException(String msg)
```

Shall not copy ”this” to any instance or static field. The scope containing the msg argument must enclose the scope containing ”this”. Otherwise, an IllegalAssignmentError will be thrown.

Allocates an application- and implementation-dependent amount of memory in the current scope (to represent stack backtrace).

B.3.21 CLASS IncompatibleClassChangeError

```
@SCJAllowed(javax.safetycritical.annotate.Level.LEVEL_2)
public class IncompatibleClassChangeError implements java.io.Serializable
    extends java.lang.RuntimeException
```

Thrown when an incompatible class change has occurred to some class definition. The definition of some class, on which the currently executing method depends, has since changed.

Constructors

```
@SCJAllowed
@SCJMaySelfSuspend(false)
@SCJMayAllocate({})
@SCJPhase({
    javax.safetycritical.annotate.Phase.STARTUP,
    javax.safetycritical.annotate.Phase.INITIALIZATION,
    javax.safetycritical.annotate.Phase.RUN,
    javax.safetycritical.annotate.Phase.CLEANUP })
public IncompatibleClassChangeError( )
```

Shall not copy ”this” to any instance or static field.

Allocates an application- and implementation-dependent amount of memory in the current scope (to represent stack backtrace).

```
@SCJAllowed
@SCJMaySelfSuspend(false)
@SCJMayAllocate({})
@SCJPhase({
    javax.safetycritical.annotate.Phase.STARTUP,
    javax.safetycritical.annotate.Phase.INITIALIZATION,
    javax.safetycritical.annotate.Phase.RUN,
    javax.safetycritical.annotate.Phase.CLEANUP })
public IncompatibleClassChangeError(String msg)
```

Shall not copy ”this” to any instance or static field. The scope containing the msg argument must enclose the scope containing ”this”. Otherwise, an IllegalAssignmentError will be thrown.

Allocates an application- and implementation-dependent amount of memory in the current scope (to represent stack backtrace).

B.3.22 CLASS IndexOutOfBoundsException

```
@SCJAllowed
public class IndexOutOfBoundsException implements java.io.Serializable extends
    java.lang.RuntimeException
```

Thrown to indicate that an index of some sort (such as to an array, to a string, or to a vector) is out of range.

Constructors

```
@SCJAllowed
@SCJMaySelfSuspend(false)
@SCJMayAllocate({})
@SCJPhase({
   javax.safetycritical.annotate.Phase.STARTUP,
   javax.safetycritical.annotate.Phase.INITIALIZATION,
   javax.safetycritical.annotate.Phase.RUN,
   javax.safetycritical.annotate.Phase.CLEANUP })
public IndexOutOfBoundsException( )
```

Shall not copy ”this” to any instance or static field.

Allocates an application- and implementation-dependent amount of memory in the current scope (to represent stack backtrace).

```
@SCJAllowed
@SCJMaySelfSuspend(false)
@SCJMayAllocate({})
@SCJPhase({
   javax.safetycritical.annotate.Phase.STARTUP,
   javax.safetycritical.annotate.Phase.INITIALIZATION,
   javax.safetycritical.annotate.Phase.RUN,
   javax.safetycritical.annotate.Phase.CLEANUP })
public IndexOutOfBoundsException(String msg)
```

Shall not copy ”this” to any instance or static field. The scope containing the msg argument must enclose the scope containing ”this”. Otherwise, an IllegalAssignmentError will be thrown.

Allocates an application- and implementation-dependent amount of memory in the current scope (to represent stack backtrace).

B.3.23 CLASS InstantiationException

```
@SCJAllowed
public class InstantiationException implements java.io.Serializable extends
    java.lang.Exception
```

Thrown when an application tries to create an instance of a class using the newInstance method in class Class, but the specified class object cannot be instantiated.

Constructors

```
@SCJAllowed
@SCJMaySelfSuspend(false)
@SCJMayAllocate({})
@SCJPhase({
   javax.safetycritical.annotate.Phase.STARTUP,
   javax.safetycritical.annotate.Phase.INITIALIZATION,
   javax.safetycritical.annotate.Phase.RUN,
   javax.safetycritical.annotate.Phase.CLEANUP })
public InstantiationException( )
```

Shall not copy ”this” to any instance or static field.

Allocates an application- and implementation-dependent amount of memory in the current scope (to represent stack backtrace).

```
@SCJAllowed
@SCJMaySelfSuspend(false)
@SCJMayAllocate({})
@SCJPhase({
   javax.safetycritical.annotate.Phase.STARTUP,
   javax.safetycritical.annotate.Phase.INITIALIZATION,
   javax.safetycritical.annotate.Phase.RUN,
   javax.safetycritical.annotate.Phase.CLEANUP })
public InstantiationException(String msg)
```

Shall not copy ”this” to any instance or static field. The scope containing the msg argument must enclose the scope containing ”this”. Otherwise, an IllegalAssignmentError will be thrown.

Allocates an application- and implementation-dependent amount of memory in the current scope (to represent stack backtrace).

B.3.24 CLASS Integer

```
@SCJAllowed
public class Integer implements java.lang.Comparable<T>, java.io.Serializable
    extends java.lang.Number
```

The Integer class wraps a value of the primitive type int in an object. An object of type Integer contains a single field whose type is int.

Fields

```
@SCJAllowed
public static final int MAX_VALUE
```

@SCJAllowed
public static final int MIN_VALUE

@SCJAllowed
public static final int SIZE

@SCJAllowed
public static final java.lang.**Class**<java.lang.Integer> TYPE

Constructors

@SCJAllowed
@SCJMaySelfSuspend(false)
@SCJMayAllocate({})
@SCJPhase({
javax.safetycritical.annotate.Phase.STARTUP,
javax.safetycritical.annotate.Phase.INITIALIZATION,
javax.safetycritical.annotate.Phase.RUN,
javax.safetycritical.annotate.Phase.CLEANUP })
public Integer(**int** val)

Allocates no memory. Does not allow ”this” to escape local variables.

@SCJAllowed
@SCJMaySelfSuspend(false)
@SCJMayAllocate({})
@SCJPhase({
javax.safetycritical.annotate.Phase.STARTUP,
javax.safetycritical.annotate.Phase.INITIALIZATION,
javax.safetycritical.annotate.Phase.RUN,
javax.safetycritical.annotate.Phase.CLEANUP })
public Integer(String str)
throws java.lang.NumberFormatException

Allocates no memory. Does not allow ”this” or ”str” argument to escape local variables.

Methods

@SCJAllowed
@SCJMaySelfSuspend(false)
@SCJMayAllocate({})
@SCJPhase({
javax.safetycritical.annotate.Phase.STARTUP,
javax.safetycritical.annotate.Phase.INITIALIZATION,

```
    javax.safetycritical.annotate.Phase.RUN,
    javax.safetycritical.annotate.Phase.CLEANUP })
public static int bitCount(int i)
```

Allocates no memory. Does not allow ”this” to escape local variables.

```
@SCJAllowed
@SCJMaySelfSuspend(false)
@SCJMayAllocate({})
@SCJPhase({
    javax.safetycritical.annotate.Phase.STARTUP,
    javax.safetycritical.annotate.Phase.INITIALIZATION,
    javax.safetycritical.annotate.Phase.RUN,
    javax.safetycritical.annotate.Phase.CLEANUP })
public byte byteValue( )
```

Allocates no memory. Does not allow ”this” to escape local variables.

```
@SCJAllowed
@SCJMaySelfSuspend(false)
@SCJMayAllocate({})
@SCJPhase({
    javax.safetycritical.annotate.Phase.STARTUP,
    javax.safetycritical.annotate.Phase.INITIALIZATION,
    javax.safetycritical.annotate.Phase.RUN,
    javax.safetycritical.annotate.Phase.CLEANUP })
public int compareTo(Integer other)
```

Allocates no memory. Does not allow ”this” or ”other” argument to escape local variables.

```
@SCJAllowed
@SCJMaySelfSuspend(false)
@SCJMayAllocate({})
@SCJPhase({
    javax.safetycritical.annotate.Phase.STARTUP,
    javax.safetycritical.annotate.Phase.INITIALIZATION,
    javax.safetycritical.annotate.Phase.RUN,
    javax.safetycritical.annotate.Phase.CLEANUP })
public static java.lang.Integer decode(String str)
  throws java.lang.NumberFormatException
```

Does not allow ”str” argument to escape local variables. Allocates an Integer in caller’s scope.

```
@SCJAllowed
@SCJMaySelfSuspend(false)
@SCJMayAllocate({})
@SCJPhase({
    javax.safetycritical.annotate.Phase.STARTUP,
    javax.safetycritical.annotate.Phase.INITIALIZATION,
    javax.safetycritical.annotate.Phase.RUN,
    javax.safetycritical.annotate.Phase.CLEANUP })
public double doubleValue( )
```

Allocates no memory.

```
@SCJAllowed
@SCJMaySelfSuspend(false)
@SCJMayAllocate({})
@SCJPhase({
    javax.safetycritical.annotate.Phase.STARTUP,
    javax.safetycritical.annotate.Phase.INITIALIZATION,
    javax.safetycritical.annotate.Phase.RUN,
    javax.safetycritical.annotate.Phase.CLEANUP })
public boolean equals(Object obj)
```

Allocates no memory. Does not allow ”this” or ”obj” argument to escape local variables.

```
@SCJAllowed
@SCJMaySelfSuspend(false)
@SCJMayAllocate({})
@SCJPhase({
    javax.safetycritical.annotate.Phase.STARTUP,
    javax.safetycritical.annotate.Phase.INITIALIZATION,
    javax.safetycritical.annotate.Phase.RUN,
    javax.safetycritical.annotate.Phase.CLEANUP })
public float floatValue( )
```

Allocates no memory. Does not allow ”this” to escape local variables.

```
@SCJAllowed
@SCJMaySelfSuspend(false)
@SCJMayAllocate({javax.safetycritical.annotate.AllocationContext.CURRENT})
@SCJPhase({
    javax.safetycritical.annotate.Phase.STARTUP,
    javax.safetycritical.annotate.Phase.INITIALIZATION,
    javax.safetycritical.annotate.Phase.RUN,
    javax.safetycritical.annotate.Phase.CLEANUP })
public static java.lang.Integer getInteger(String str, Integer v)
```

Does not allow ”str” or ”v” arguments to escape local variables. Allocates Integer in caller’s scope.

```
@SCJAllowed
@SCJMaySelfSuspend(false)
@SCJMayAllocate({javax.safetycritical.annotate.AllocationContext.CURRENT})
@SCJPhase({
   javax.safetycritical.annotate.Phase.STARTUP,
   javax.safetycritical.annotate.Phase.INITIALIZATION,
   javax.safetycritical.annotate.Phase.RUN,
   javax.safetycritical.annotate.Phase.CLEANUP })
public static java.lang.Integer getInteger(String str, int v)
```

Does not allow ”str” argument to escape local variables. Allocates Integer in caller’s scope.

```
@SCJAllowed
@SCJMaySelfSuspend(false)
@SCJMayAllocate({javax.safetycritical.annotate.AllocationContext.CURRENT})
@SCJPhase({
   javax.safetycritical.annotate.Phase.STARTUP,
   javax.safetycritical.annotate.Phase.INITIALIZATION,
   javax.safetycritical.annotate.Phase.RUN,
   javax.safetycritical.annotate.Phase.CLEANUP })
public static java.lang.Integer getInteger(String str)
```

Does not allow ”str” argument to escape local variables. Allocates Integer in caller’s scope.

```
@SCJAllowed
@SCJMaySelfSuspend(false)
@SCJMayAllocate({})
@SCJPhase({
   javax.safetycritical.annotate.Phase.STARTUP,
   javax.safetycritical.annotate.Phase.INITIALIZATION,
   javax.safetycritical.annotate.Phase.RUN,
   javax.safetycritical.annotate.Phase.CLEANUP })
public int hashCode( )
```

Allocates no memory. Does not allow ”this” to escape local variables.

```
@SCJAllowed
@SCJMaySelfSuspend(false)
@SCJMayAllocate({})
```

```
@SCJPhase({
    javax.safetycritical.annotate.Phase.STARTUP,
    javax.safetycritical.annotate.Phase.INITIALIZATION,
    javax.safetycritical.annotate.Phase.RUN,
    javax.safetycritical.annotate.Phase.CLEANUP })
public static int highestOneBit(int i)
```

Allocates no memory.

```
@SCJAllowed
@SCJMaySelfSuspend(false)
@SCJMayAllocate({})
@SCJPhase({
    javax.safetycritical.annotate.Phase.STARTUP,
    javax.safetycritical.annotate.Phase.INITIALIZATION,
    javax.safetycritical.annotate.Phase.RUN,
    javax.safetycritical.annotate.Phase.CLEANUP })
public int intValue( )
```

Allocates no memory. Does not allow ”this” to escape local variables.

```
@SCJAllowed
@SCJMaySelfSuspend(false)
@SCJMayAllocate({})
@SCJPhase({
    javax.safetycritical.annotate.Phase.STARTUP,
    javax.safetycritical.annotate.Phase.INITIALIZATION,
    javax.safetycritical.annotate.Phase.RUN,
    javax.safetycritical.annotate.Phase.CLEANUP })
public long longValue( )
```

Allocates no memory. Does not allow ”this” to escape local variables.

```
@SCJAllowed
@SCJMaySelfSuspend(false)
@SCJMayAllocate({})
@SCJPhase({
    javax.safetycritical.annotate.Phase.STARTUP,
    javax.safetycritical.annotate.Phase.INITIALIZATION,
    javax.safetycritical.annotate.Phase.RUN,
    javax.safetycritical.annotate.Phase.CLEANUP })
public static int lowestOneBit(int i)
```

Allocates no memory.

```
@SCJAllowed
@SCJMaySelfSuspend(false)
@SCJMayAllocate({})
@SCJPhase({
    javax.safetycritical.annotate.Phase.STARTUP,
    javax.safetycritical.annotate.Phase.INITIALIZATION,
    javax.safetycritical.annotate.Phase.RUN,
    javax.safetycritical.annotate.Phase.CLEANUP })
public static int numberOfLeadingZeros(int i)
```

Allocates no memory.

```
@SCJAllowed
@SCJMaySelfSuspend(false)
@SCJMayAllocate({})
@SCJPhase({
    javax.safetycritical.annotate.Phase.STARTUP,
    javax.safetycritical.annotate.Phase.INITIALIZATION,
    javax.safetycritical.annotate.Phase.RUN,
    javax.safetycritical.annotate.Phase.CLEANUP })
public static int parseInt(String str, int radix)
  throws java.lang.NumberFormatException
```

Allocates no memory. Does not allow ”str” argument to escape local variables.

```
@SCJAllowed
@SCJMaySelfSuspend(false)
@SCJMayAllocate({})
@SCJPhase({
    javax.safetycritical.annotate.Phase.STARTUP,
    javax.safetycritical.annotate.Phase.INITIALIZATION,
    javax.safetycritical.annotate.Phase.RUN,
    javax.safetycritical.annotate.Phase.CLEANUP })
public static int parseInt(String str)
  throws java.lang.NumberFormatException
```

Allocates no memory. Does not allow ”str” argument to escape local variables.

```
@SCJAllowed
@SCJMaySelfSuspend(false)
@SCJMayAllocate({})
@SCJPhase({
    javax.safetycritical.annotate.Phase.STARTUP,
    javax.safetycritical.annotate.Phase.INITIALIZATION,
    javax.safetycritical.annotate.Phase.RUN,
    javax.safetycritical.annotate.Phase.CLEANUP })
public static int reverse(int i)
```

Allocates no memory.

```
@SCJAllowed
@SCJMaySelfSuspend(false)
@SCJMayAllocate({})
@SCJPhase({
    javax.safetycritical.annotate.Phase.STARTUP,
    javax.safetycritical.annotate.Phase.INITIALIZATION,
    javax.safetycritical.annotate.Phase.RUN,
    javax.safetycritical.annotate.Phase.CLEANUP })
public static int reverseBytes(int i)
```

Allocates no memory.

```
@SCJAllowed
@SCJMaySelfSuspend(false)
@SCJMayAllocate({})
@SCJPhase({
    javax.safetycritical.annotate.Phase.STARTUP,
    javax.safetycritical.annotate.Phase.INITIALIZATION,
    javax.safetycritical.annotate.Phase.RUN,
    javax.safetycritical.annotate.Phase.CLEANUP })
public static int rotateLeft(int i, int distance)
```

Allocates no memory.

```
@SCJAllowed
@SCJMaySelfSuspend(false)
@SCJMayAllocate({})
@SCJPhase({
    javax.safetycritical.annotate.Phase.STARTUP,
    javax.safetycritical.annotate.Phase.INITIALIZATION,
    javax.safetycritical.annotate.Phase.RUN,
    javax.safetycritical.annotate.Phase.CLEANUP })
public static int rotateRight(int i, int distance)
```

Allocates no memory.

```
@SCJAllowed
@SCJMaySelfSuspend(false)
@SCJMayAllocate({})
@SCJPhase({
    javax.safetycritical.annotate.Phase.STARTUP,
    javax.safetycritical.annotate.Phase.INITIALIZATION,
    javax.safetycritical.annotate.Phase.RUN,
    javax.safetycritical.annotate.Phase.CLEANUP })
public short shortValue( )
```

Allocates no memory. Does not allow ”this” to escape local variables.

```
@SCJAllowed
@SCJMaySelfSuspend(false)
@SCJMayAllocate({})
@SCJPhase({
   javax.safetycritical.annotate.Phase.STARTUP,
   javax.safetycritical.annotate.Phase.INITIALIZATION,
   javax.safetycritical.annotate.Phase.RUN,
   javax.safetycritical.annotate.Phase.CLEANUP })
public static int sigNum(int i)
```

Allocates no memory.

```
@SCJAllowed
@SCJMaySelfSuspend(false)
@SCJMayAllocate({javax.safetycritical.annotate.AllocationContext.CURRENT})
@SCJPhase({
   javax.safetycritical.annotate.Phase.STARTUP,
   javax.safetycritical.annotate.Phase.INITIALIZATION,
   javax.safetycritical.annotate.Phase.RUN,
   javax.safetycritical.annotate.Phase.CLEANUP })
public static java.lang.String toBinaryString(int v)
```

Does not allow ”this” to escape local variables. Allocates a String and associated internal ”structure” (e.g. char[]) in caller’s scope.

```
@SCJAllowed
@SCJMaySelfSuspend(false)
@SCJMayAllocate({javax.safetycritical.annotate.AllocationContext.CURRENT})
@SCJPhase({
   javax.safetycritical.annotate.Phase.STARTUP,
   javax.safetycritical.annotate.Phase.INITIALIZATION,
   javax.safetycritical.annotate.Phase.RUN,
   javax.safetycritical.annotate.Phase.CLEANUP })
public static java.lang.String toHexString(int v)
```

Does not allow ”this” to escape local variables. Allocates a String and associated internal ”structure” (e.g. char[]) in caller’s scope.

```
@SCJAllowed
@SCJMaySelfSuspend(false)
@SCJMayAllocate({javax.safetycritical.annotate.AllocationContext.CURRENT})
@SCJPhase({
```

```
    javax.safetycritical.annotate.Phase.STARTUP,
    javax.safetycritical.annotate.Phase.INITIALIZATION,
    javax.safetycritical.annotate.Phase.RUN,
    javax.safetycritical.annotate.Phase.CLEANUP })
public static java.lang.String toOctalString(int v)
```

Does not allow ”this” to escape local variables. Allocates a String and associated internal ”structure” (e.g. char[]) in caller’s scope.

```
@SCJAllowed
@SCJMaySelfSuspend(false)
@SCJMayAllocate({javax.safetycritical.annotate.AllocationContext.CURRENT})
@SCJPhase({
    javax.safetycritical.annotate.Phase.STARTUP,
    javax.safetycritical.annotate.Phase.INITIALIZATION,
    javax.safetycritical.annotate.Phase.RUN,
    javax.safetycritical.annotate.Phase.CLEANUP })
public static java.lang.String toString(int v)
```

Allocates a String and associated internal ”structure” (e.g. char[]) in caller’s scope.

```
@SCJAllowed
@SCJMaySelfSuspend(false)
@SCJMayAllocate({javax.safetycritical.annotate.AllocationContext.CURRENT})
@SCJPhase({
    javax.safetycritical.annotate.Phase.STARTUP,
    javax.safetycritical.annotate.Phase.INITIALIZATION,
    javax.safetycritical.annotate.Phase.RUN,
    javax.safetycritical.annotate.Phase.CLEANUP })
public static java.lang.String toString(int v, int base)
```

Does not allow ”this” to escape local variables. Allocates a String and associated internal ”structure” (e.g. char[]) in caller’s scope.

```
@SCJAllowed
@SCJMaySelfSuspend(false)
@SCJMayAllocate({javax.safetycritical.annotate.AllocationContext.CURRENT})
@SCJPhase({
    javax.safetycritical.annotate.Phase.STARTUP,
    javax.safetycritical.annotate.Phase.INITIALIZATION,
    javax.safetycritical.annotate.Phase.RUN,
    javax.safetycritical.annotate.Phase.CLEANUP })
public java.lang.String toString( )
```

Does not allow ”this” to escape local variables. Allocates a String and associated internal ”structure” (e.g. char[]) in caller’s scope. (Note: this semantics is desired for consistency with overridden implementation of Object.toString()).

```
@SCJAllowed
@SCJMaySelfSuspend(false)
@SCJMayAllocate({javax.safetycritical.annotate.AllocationContext.CURRENT})
@SCJPhase({
   javax.safetycritical.annotate.Phase.STARTUP,
   javax.safetycritical.annotate.Phase.INITIALIZATION,
   javax.safetycritical.annotate.Phase.RUN,
   javax.safetycritical.annotate.Phase.CLEANUP })
public static java.lang.Integer valueOf(String str, int base)
  throws java.lang.NumberFormatException
```

Does not allow ”str” argument to escape local variables. Allocates an Integer in caller’s scope.

```
@SCJAllowed
@SCJMaySelfSuspend(false)
@SCJMayAllocate({javax.safetycritical.annotate.AllocationContext.CURRENT})
@SCJPhase({
   javax.safetycritical.annotate.Phase.STARTUP,
   javax.safetycritical.annotate.Phase.INITIALIZATION,
   javax.safetycritical.annotate.Phase.RUN,
   javax.safetycritical.annotate.Phase.CLEANUP })
public static java.lang.Integer valueOf(String str)
  throws java.lang.NumberFormatException
```

Does not allow ”str” argument to escape local variables. Allocates an Integer in caller’s scope.

```
@SCJAllowed
@SCJMaySelfSuspend(false)
@SCJMayAllocate({javax.safetycritical.annotate.AllocationContext.CURRENT})
@SCJPhase({
   javax.safetycritical.annotate.Phase.STARTUP,
   javax.safetycritical.annotate.Phase.INITIALIZATION,
   javax.safetycritical.annotate.Phase.RUN,
   javax.safetycritical.annotate.Phase.CLEANUP })
public static java.lang.Integer valueOf(int val)
```

Allocates an Integer in caller’s scope.

B.3.25 CLASS InternalError

```
@SCJAllowed
public class InternalError implements java.io.Serializable extends
    java.lang.VirtualMachineError
```

Thrown to indicate some unexpected internal error has occurred in the Java Virtual Machine.

Constructors

```
@SCJAllowed
@SCJMaySelfSuspend(false)
@SCJMayAllocate({})
@SCJPhase({
   javax.safetycritical.annotate.Phase.STARTUP,
   javax.safetycritical.annotate.Phase.INITIALIZATION,
   javax.safetycritical.annotate.Phase.RUN,
   javax.safetycritical.annotate.Phase.CLEANUP })
public InternalError( )
```

Shall not copy ”this” to any instance or static field.

Allocates an application- and implementation-dependent amount of memory in the current scope (to represent stack backtrace).

```
@SCJAllowed
@SCJMaySelfSuspend(false)
@SCJMayAllocate({})
@SCJPhase({
   javax.safetycritical.annotate.Phase.STARTUP,
   javax.safetycritical.annotate.Phase.INITIALIZATION,
   javax.safetycritical.annotate.Phase.RUN,
   javax.safetycritical.annotate.Phase.CLEANUP })
public InternalError(String msg)
```

Shall not copy ”this” to any instance or static field. The scope containing the msg argument must enclose the scope containing ”this”. Otherwise, an IllegalAssignmentError will be thrown.

Allocates an application- and implementation-dependent amount of memory in the current scope (to represent stack backtrace).

B.3.26 CLASS InterruptedException

```
@SCJAllowed
public class InterruptedException implements java.io.Serializable extends
    java.lang.Exception
```

Thrown when a thread is waiting, sleeping, or otherwise occupied, and the thread is interrupted, either before or during the activity.

Constructors

```
@SCJAllowed
@SCJMaySelfSuspend(false)
@SCJMayAllocate({})
@SCJPhase({
   javax.safetycritical.annotate.Phase.STARTUP,
   javax.safetycritical.annotate.Phase.INITIALIZATION,
   javax.safetycritical.annotate.Phase.RUN,
   javax.safetycritical.annotate.Phase.CLEANUP })
public InterruptedException( )
```

Shall not copy ”this” to any instance or static field.

Allocates an application- and implementation-dependent amount of memory in the current scope (to represent stack backtrace).

```
@SCJAllowed
@SCJMaySelfSuspend(false)
@SCJMayAllocate({})
@SCJPhase({
   javax.safetycritical.annotate.Phase.STARTUP,
   javax.safetycritical.annotate.Phase.INITIALIZATION,
   javax.safetycritical.annotate.Phase.RUN,
   javax.safetycritical.annotate.Phase.CLEANUP })
public InterruptedException(String msg)
```

Shall not copy ”this” to any instance or static field. The scope containing the msg argument must enclose the scope containing ”this”. Otherwise, an IllegalAssignmentError will be thrown.

Allocates an application- and implementation-dependent amount of memory in the current scope (to represent stack backtrace).

B.3.27 CLASS Long

@SCJAllowed
public class Long implements java.lang.Comparable<T>, java.io.Serializable
extends java.lang.Number

The Long class wraps a value of the primitive type long in an object. An object of type Long contains a single field whose type is long.

Fields

@SCJAllowed
public static final long MAX_VALUE

@SCJAllowed
public static final long MIN_VALUE

@SCJAllowed
public static final int SIZE

@SCJAllowed
public static final java.lang.**Class**<java.lang.**Long**> TYPE

Constructors

```
@SCJAllowed
@SCJMaySelfSuspend(false)
@SCJMayAllocate({
   javax.safetycritical.annotate.AllocationContext.CURRENT,
   javax.safetycritical.annotate.AllocationContext.INNER,
   javax.safetycritical.annotate.AllocationContext.OUTER})
@SCJPhase({
   javax.safetycritical.annotate.Phase.STARTUP,
   javax.safetycritical.annotate.Phase.INITIALIZATION,
   javax.safetycritical.annotate.Phase.RUN,
   javax.safetycritical.annotate.Phase.CLEANUP })
public Long(long val)
```

Allocates no memory. Does not allow ”this” to escape local variables.

```
@SCJAllowed
@SCJMaySelfSuspend(false)
@SCJMayAllocate({
   javax.safetycritical.annotate.AllocationContext.CURRENT,
```

```
    javax.safetycritical.annotate.AllocationContext.INNER,
    javax.safetycritical.annotate.AllocationContext.OUTER})
@SCJPhase({
    javax.safetycritical.annotate.Phase.STARTUP,
    javax.safetycritical.annotate.Phase.INITIALIZATION,
    javax.safetycritical.annotate.Phase.RUN,
    javax.safetycritical.annotate.Phase.CLEANUP })
public Long(String str)
  throws java.lang.NumberFormatException
```

Allocates no memory. Does not allow ”this” or ”str” argument to escape local variables.

Throws NumberFormatException

Methods

```
@SCJAllowed
@SCJMaySelfSuspend(false)
@SCJMayAllocate({})
@SCJPhase({
    javax.safetycritical.annotate.Phase.STARTUP,
    javax.safetycritical.annotate.Phase.INITIALIZATION,
    javax.safetycritical.annotate.Phase.RUN,
    javax.safetycritical.annotate.Phase.CLEANUP })
public static int bitCount(long i)
```

Allocates no memory.

```
@SCJAllowed
@SCJMaySelfSuspend(false)
@SCJMayAllocate({})
@SCJPhase({
    javax.safetycritical.annotate.Phase.STARTUP,
    javax.safetycritical.annotate.Phase.INITIALIZATION,
    javax.safetycritical.annotate.Phase.RUN,
    javax.safetycritical.annotate.Phase.CLEANUP })
public byte byteValue( )
```

Allocates no memory. Does not allow ”this” to escape local variables.

```
@SCJAllowed
@SCJMaySelfSuspend(false)
@SCJMayAllocate({})
@SCJPhase({
    javax.safetycritical.annotate.Phase.STARTUP,
```

```
    javax.safetycritical.annotate.Phase.INITIALIZATION,
    javax.safetycritical.annotate.Phase.RUN,
    javax.safetycritical.annotate.Phase.CLEANUP })
public int compareTo(Long other)
```

Allocates no memory. Does not allow ”this” or ”other” argument to escape local variables.

```
@SCJAllowed
@SCJMaySelfSuspend(false)
@SCJMayAllocate({
    javax.safetycritical.annotate.AllocationContext.CURRENT,
    javax.safetycritical.annotate.AllocationContext.INNER,
    javax.safetycritical.annotate.AllocationContext.OUTER})
@SCJPhase({
    javax.safetycritical.annotate.Phase.STARTUP,
    javax.safetycritical.annotate.Phase.INITIALIZATION,
    javax.safetycritical.annotate.Phase.RUN,
    javax.safetycritical.annotate.Phase.CLEANUP })
public static java.lang.Long decode(String str)
  throws java.lang.NumberFormatException
```

Does not allow ”str” argument to escape local variables. Allocates a Long result object in the caller’s scope.

Throws NumberFormatException

```
@SCJAllowed
@SCJMaySelfSuspend(false)
@SCJMayAllocate({})
@SCJPhase({
    javax.safetycritical.annotate.Phase.STARTUP,
    javax.safetycritical.annotate.Phase.INITIALIZATION,
    javax.safetycritical.annotate.Phase.RUN,
    javax.safetycritical.annotate.Phase.CLEANUP })
public double doubleValue( )
```

Allocates no memory. Does not allow ”this” to escape local variables.

```
@SCJAllowed
@SCJMaySelfSuspend(false)
@SCJMayAllocate({})
@SCJPhase({
    javax.safetycritical.annotate.Phase.STARTUP,
    javax.safetycritical.annotate.Phase.INITIALIZATION,
    javax.safetycritical.annotate.Phase.RUN,
    javax.safetycritical.annotate.Phase.CLEANUP })
public boolean equals(Object obj)
```

Allocates no memory. Does not allow ”this” or ”obj” argument to escape local variables.

```
@SCJAllowed
@SCJMaySelfSuspend(false)
@SCJMayAllocate({})
@SCJPhase({
    javax.safetycritical.annotate.Phase.STARTUP,
    javax.safetycritical.annotate.Phase.INITIALIZATION,
    javax.safetycritical.annotate.Phase.RUN,
    javax.safetycritical.annotate.Phase.CLEANUP })
public float floatValue( )
```

Allocates no memory. Does not allow ”this” to escape local variables.

```
@SCJAllowed
@SCJMaySelfSuspend(false)
@SCJMayAllocate({
    javax.safetycritical.annotate.AllocationContext.CURRENT,
    javax.safetycritical.annotate.AllocationContext.INNER,
    javax.safetycritical.annotate.AllocationContext.OUTER})
@SCJPhase({
    javax.safetycritical.annotate.Phase.STARTUP,
    javax.safetycritical.annotate.Phase.INITIALIZATION,
    javax.safetycritical.annotate.Phase.RUN,
    javax.safetycritical.annotate.Phase.CLEANUP })
public static java.lang.Long getLong(String str, Long v)
```

Does not allow ”str” argument to escape local variables. Allocates a Long result object in the caller’s scope.

```
@SCJAllowed
@SCJMaySelfSuspend(false)
@SCJMayAllocate({
    javax.safetycritical.annotate.AllocationContext.CURRENT,
    javax.safetycritical.annotate.AllocationContext.INNER,
    javax.safetycritical.annotate.AllocationContext.OUTER})
@SCJPhase({
    javax.safetycritical.annotate.Phase.STARTUP,
    javax.safetycritical.annotate.Phase.INITIALIZATION,
    javax.safetycritical.annotate.Phase.RUN,
    javax.safetycritical.annotate.Phase.CLEANUP })
public static java.lang.Long getLong(String str, long v)
```

Does not allow ”str” argument to escape local variables. Allocates a Long result object in the caller’s scope.

```
@SCJAllowed
@SCJMaySelfSuspend(false)
@SCJMayAllocate({
   javax.safetycritical.annotate.AllocationContext.CURRENT,
   javax.safetycritical.annotate.AllocationContext.INNER,
   javax.safetycritical.annotate.AllocationContext.OUTER})
@SCJPhase({
   javax.safetycritical.annotate.Phase.STARTUP,
   javax.safetycritical.annotate.Phase.INITIALIZATION,
   javax.safetycritical.annotate.Phase.RUN,
   javax.safetycritical.annotate.Phase.CLEANUP })
public static java.lang.Long getLong(String str)
```

Does not allow ”str” argument to escape local variables. Allocates a Long result object in the caller’s scope.

```
@SCJAllowed
@SCJMaySelfSuspend(false)
@SCJMayAllocate({})
@SCJPhase({
   javax.safetycritical.annotate.Phase.STARTUP,
   javax.safetycritical.annotate.Phase.INITIALIZATION,
   javax.safetycritical.annotate.Phase.RUN,
   javax.safetycritical.annotate.Phase.CLEANUP })
public int hashCode( )
```

Allocates no memory. Does not allow ”this” to escape local variables.

```
@SCJAllowed
@SCJMaySelfSuspend(false)
@SCJMayAllocate({})
@SCJPhase({
   javax.safetycritical.annotate.Phase.STARTUP,
   javax.safetycritical.annotate.Phase.INITIALIZATION,
   javax.safetycritical.annotate.Phase.RUN,
   javax.safetycritical.annotate.Phase.CLEANUP })
public static long highestOneBit(long i)
```

Allocates no memory.

```
@SCJAllowed
@SCJMaySelfSuspend(false)
@SCJMayAllocate({})
@SCJPhase({
   javax.safetycritical.annotate.Phase.STARTUP,
   javax.safetycritical.annotate.Phase.INITIALIZATION,
```

```
    javax.safetycritical.annotate.Phase.RUN,
    javax.safetycritical.annotate.Phase.CLEANUP })
public int intValue( )
```

Allocates no memory. Does not allow ”this” to escape local variables.

```
@SCJAllowed
@SCJMaySelfSuspend(false)
@SCJMayAllocate({})
@SCJPhase({
    javax.safetycritical.annotate.Phase.STARTUP,
    javax.safetycritical.annotate.Phase.INITIALIZATION,
    javax.safetycritical.annotate.Phase.RUN,
    javax.safetycritical.annotate.Phase.CLEANUP })
public long longValue( )
```

Allocates no memory. Does not allow ”this” to escape local variables.

```
@SCJAllowed
@SCJMaySelfSuspend(false)
@SCJMayAllocate({})
@SCJPhase({
    javax.safetycritical.annotate.Phase.STARTUP,
    javax.safetycritical.annotate.Phase.INITIALIZATION,
    javax.safetycritical.annotate.Phase.RUN,
    javax.safetycritical.annotate.Phase.CLEANUP })
public static long lowestOneBit(long i)
```

Allocates no memory.

```
@SCJAllowed
@SCJMaySelfSuspend(false)
@SCJMayAllocate({})
@SCJPhase({
    javax.safetycritical.annotate.Phase.STARTUP,
    javax.safetycritical.annotate.Phase.INITIALIZATION,
    javax.safetycritical.annotate.Phase.RUN,
    javax.safetycritical.annotate.Phase.CLEANUP })
public static int numberOfLeadingZeros(long i)
```

Allocates no memory.

```
@SCJAllowed
@SCJMaySelfSuspend(false)
@SCJMayAllocate({})
@SCJPhase({
    javax.safetycritical.annotate.Phase.STARTUP,
    javax.safetycritical.annotate.Phase.INITIALIZATION,
    javax.safetycritical.annotate.Phase.RUN,
    javax.safetycritical.annotate.Phase.CLEANUP })
public static int numberOfTrailingZeros(long i)
```

Allocates no memory.

```
@SCJAllowed
@SCJMaySelfSuspend(false)
@SCJMayAllocate({})
@SCJPhase({
    javax.safetycritical.annotate.Phase.STARTUP,
    javax.safetycritical.annotate.Phase.INITIALIZATION,
    javax.safetycritical.annotate.Phase.RUN,
    javax.safetycritical.annotate.Phase.CLEANUP })
public static long parseLong(String str, int base)
  throws java.lang.NumberFormatException
```

Allocates no memory. Does not allow ”this” or ”other” argument to escape local variables.

```
@SCJAllowed
@SCJMaySelfSuspend(false)
@SCJMayAllocate({})
@SCJPhase({
    javax.safetycritical.annotate.Phase.STARTUP,
    javax.safetycritical.annotate.Phase.INITIALIZATION,
    javax.safetycritical.annotate.Phase.RUN,
    javax.safetycritical.annotate.Phase.CLEANUP })
public static long parseLong(String str)
  throws java.lang.NumberFormatException
```

Allocates no memory. Does not allow ”this” or ”other” argument to escape local variables.

Throws NumberFormatException

```
@SCJAllowed
@SCJMaySelfSuspend(false)
@SCJMayAllocate({})
@SCJPhase({
```

```
    javax.safetycritical.annotate.Phase.STARTUP,
    javax.safetycritical.annotate.Phase.INITIALIZATION,
    javax.safetycritical.annotate.Phase.RUN,
    javax.safetycritical.annotate.Phase.CLEANUP })
public static long reverse(long i)
```

Allocates no memory.

```
@SCJAllowed
@SCJMaySelfSuspend(false)
@SCJMayAllocate({})
@SCJPhase({
    javax.safetycritical.annotate.Phase.STARTUP,
    javax.safetycritical.annotate.Phase.INITIALIZATION,
    javax.safetycritical.annotate.Phase.RUN,
    javax.safetycritical.annotate.Phase.CLEANUP })
public static long reverseBytes(long i)
```

Allocates no memory.

```
@SCJAllowed
@SCJMaySelfSuspend(false)
@SCJMayAllocate({})
@SCJPhase({
    javax.safetycritical.annotate.Phase.STARTUP,
    javax.safetycritical.annotate.Phase.INITIALIZATION,
    javax.safetycritical.annotate.Phase.RUN,
    javax.safetycritical.annotate.Phase.CLEANUP })
public static long rotateLeft(long i, int distance)
```

Allocates no memory.

```
@SCJAllowed
@SCJMaySelfSuspend(false)
@SCJMayAllocate({})
@SCJPhase({
    javax.safetycritical.annotate.Phase.STARTUP,
    javax.safetycritical.annotate.Phase.INITIALIZATION,
    javax.safetycritical.annotate.Phase.RUN,
    javax.safetycritical.annotate.Phase.CLEANUP })
public static long rotateRight(long i, int distance)
```

Allocates no memory.

```
@SCJAllowed
@SCJMaySelfSuspend(false)
@SCJMayAllocate({})
@SCJPhase({
   javax.safetycritical.annotate.Phase.STARTUP,
   javax.safetycritical.annotate.Phase.INITIALIZATION,
   javax.safetycritical.annotate.Phase.RUN,
   javax.safetycritical.annotate.Phase.CLEANUP })
public short shortValue( )
```

Allocates no memory. Does not allow ”this” to escape local variables.

```
@SCJAllowed
@SCJMaySelfSuspend(false)
@SCJMayAllocate({})
@SCJPhase({
   javax.safetycritical.annotate.Phase.STARTUP,
   javax.safetycritical.annotate.Phase.INITIALIZATION,
   javax.safetycritical.annotate.Phase.RUN,
   javax.safetycritical.annotate.Phase.CLEANUP })
public static int signum(long i)
```

Allocates no memory.

```
@SCJAllowed
@SCJMaySelfSuspend(false)
@SCJMayAllocate({javax.safetycritical.annotate.AllocationContext.CURRENT})
@SCJPhase({
   javax.safetycritical.annotate.Phase.STARTUP,
   javax.safetycritical.annotate.Phase.INITIALIZATION,
   javax.safetycritical.annotate.Phase.RUN,
   javax.safetycritical.annotate.Phase.CLEANUP })
public static java.lang.String toBinaryString(long v)
```

Does not allow ”this” to escape local variables. Allocates a String and associated internal ”structure” (e.g. char[]) in caller’s scope.

```
@SCJAllowed
@SCJMaySelfSuspend(false)
@SCJMayAllocate({javax.safetycritical.annotate.AllocationContext.CURRENT})
@SCJPhase({
   javax.safetycritical.annotate.Phase.STARTUP,
   javax.safetycritical.annotate.Phase.INITIALIZATION,
   javax.safetycritical.annotate.Phase.RUN,
   javax.safetycritical.annotate.Phase.CLEANUP })
public static java.lang.String toHexString(long v)
```

Does not allow ”this” to escape local variables. Allocates a String and associated internal ”structure” (e.g. char[]) in caller’s scope.

```
@SCJAllowed
@SCJMaySelfSuspend(false)
@SCJMayAllocate({javax.safetycritical.annotate.AllocationContext.CURRENT})
@SCJPhase({
   javax.safetycritical.annotate.Phase.STARTUP,
   javax.safetycritical.annotate.Phase.INITIALIZATION,
   javax.safetycritical.annotate.Phase.RUN,
   javax.safetycritical.annotate.Phase.CLEANUP })
public static java.lang.String toOctalString(long v)
```

Does not allow ”this” to escape local variables. Allocates a String and associated internal ”structure” (e.g. char[]) in caller’s scope.

```
@SCJAllowed
@SCJMaySelfSuspend(false)
@SCJMayAllocate({javax.safetycritical.annotate.AllocationContext.CURRENT})
@SCJPhase({
   javax.safetycritical.annotate.Phase.STARTUP,
   javax.safetycritical.annotate.Phase.INITIALIZATION,
   javax.safetycritical.annotate.Phase.RUN,
   javax.safetycritical.annotate.Phase.CLEANUP })
public static java.lang.String toString(long v)
```

Allocates a String and associated internal ”structure” (e.g. char[]) in caller’s scope.

```
@SCJAllowed
@SCJMaySelfSuspend(false)
@SCJMayAllocate({javax.safetycritical.annotate.AllocationContext.CURRENT})
@SCJPhase({
   javax.safetycritical.annotate.Phase.STARTUP,
   javax.safetycritical.annotate.Phase.INITIALIZATION,
   javax.safetycritical.annotate.Phase.RUN,
   javax.safetycritical.annotate.Phase.CLEANUP })
public static java.lang.String toString(long v, int base)
```

Allocates a String and associated internal ”structure” (e.g. char[]) in caller’s scope.

```
@SCJAllowed
@SCJMaySelfSuspend(false)
@SCJMayAllocate({javax.safetycritical.annotate.AllocationContext.CURRENT})
@SCJPhase({
   javax.safetycritical.annotate.Phase.STARTUP,
   javax.safetycritical.annotate.Phase.INITIALIZATION,
   javax.safetycritical.annotate.Phase.RUN,
   javax.safetycritical.annotate.Phase.CLEANUP })
public java.lang.String toString( )
```

Does not allow ”this” to escape local variables. Allocates a String and associated internal ”structure” (e.g. char[]) in caller’s scope. (Note: this semantics is desired for consistency with overridden implementation of Object.toString()).

```
@SCJAllowed
@SCJMaySelfSuspend(false)
@SCJMayAllocate({javax.safetycritical.annotate.AllocationContext.CURRENT})
@SCJPhase({
   javax.safetycritical.annotate.Phase.STARTUP,
   javax.safetycritical.annotate.Phase.INITIALIZATION,
   javax.safetycritical.annotate.Phase.RUN,
   javax.safetycritical.annotate.Phase.CLEANUP })
public static java.lang.Long valueOf(String str, int base)
  throws java.lang.NumberFormatException
```

Does not allow ”str” argument to escape local variables. Allocates a Long in caller’s scope.

Throws NumberFormatException

```
@SCJAllowed
@SCJMaySelfSuspend(false)
@SCJMayAllocate({javax.safetycritical.annotate.AllocationContext.CURRENT})
@SCJPhase({
   javax.safetycritical.annotate.Phase.STARTUP,
   javax.safetycritical.annotate.Phase.INITIALIZATION,
   javax.safetycritical.annotate.Phase.RUN,
   javax.safetycritical.annotate.Phase.CLEANUP })
public static java.lang.Long valueOf(String str)
  throws java.lang.NumberFormatException
```

Does not allow ”str” argument to escape local variables. Allocates a Long in caller’s scope.

Throws NumberFormatException

```
@SCJAllowed
@SCJMaySelfSuspend(false)
@SCJMayAllocate({javax.safetycritical.annotate.AllocationContext.CURRENT})
@SCJPhase({
   javax.safetycritical.annotate.Phase.STARTUP,
   javax.safetycritical.annotate.Phase.INITIALIZATION,
   javax.safetycritical.annotate.Phase.RUN,
   javax.safetycritical.annotate.Phase.CLEANUP })
public static java.lang.Long valueOf(long val)
```

Allocates a Long in caller's scope.

B.3.28 CLASS Math

```
@SCJAllowed
public final class Math extends java.lang.Object
```

The class Math contains methods for performing basic numeric operations such as the elementary exponential, logarithm, square root, and trigonometric functions.

Fields

```
@SCJAllowed
public static final double E
```

```
@SCJAllowed
public static final double PI
```

Constructors

```
@SCJAllowed
@SCJMayAllocate({
   javax.safetycritical.annotate.AllocationContext.CURRENT,
   javax.safetycritical.annotate.AllocationContext.INNER,
   javax.safetycritical.annotate.AllocationContext.OUTER})
@SCJMaySelfSuspend(true)
@SCJPhase({
   javax.safetycritical.annotate.Phase.STARTUP,
   javax.safetycritical.annotate.Phase.INITIALIZATION,
   javax.safetycritical.annotate.Phase.RUN,
   javax.safetycritical.annotate.Phase.CLEANUP })
public Math( )
```

Methods

```
@SCJAllowed
@SCJMaySelfSuspend(false)
@SCJMayAllocate({})
@SCJPhase({
    javax.safetycritical.annotate.Phase.STARTUP,
    javax.safetycritical.annotate.Phase.INITIALIZATION,
    javax.safetycritical.annotate.Phase.RUN,
    javax.safetycritical.annotate.Phase.CLEANUP })
public static double IEEEremainder(double f1, double f2)
```

Allocates no memory.

```
@SCJAllowed
@SCJMaySelfSuspend(false)
@SCJMayAllocate({})
@SCJPhase({
    javax.safetycritical.annotate.Phase.STARTUP,
    javax.safetycritical.annotate.Phase.INITIALIZATION,
    javax.safetycritical.annotate.Phase.RUN,
    javax.safetycritical.annotate.Phase.CLEANUP })
public static long abs(long a)
```

Allocates no memory.

```
@SCJAllowed
@SCJMaySelfSuspend(false)
@SCJMayAllocate({})
@SCJPhase({
    javax.safetycritical.annotate.Phase.STARTUP,
    javax.safetycritical.annotate.Phase.INITIALIZATION,
    javax.safetycritical.annotate.Phase.RUN,
    javax.safetycritical.annotate.Phase.CLEANUP })
public static double abs(double a)
```

Allocates no memory.

```
@SCJAllowed
@SCJMaySelfSuspend(false)
@SCJMayAllocate({})
@SCJPhase({
    javax.safetycritical.annotate.Phase.STARTUP,
    javax.safetycritical.annotate.Phase.INITIALIZATION,
    javax.safetycritical.annotate.Phase.RUN,
    javax.safetycritical.annotate.Phase.CLEANUP })
public static float abs(float a)
```

Allocates no memory.

```
@SCJAllowed
@SCJMaySelfSuspend(false)
@SCJMayAllocate({})
@SCJPhase({
    javax.safetycritical.annotate.Phase.STARTUP,
    javax.safetycritical.annotate.Phase.INITIALIZATION,
    javax.safetycritical.annotate.Phase.RUN,
    javax.safetycritical.annotate.Phase.CLEANUP })
public static int abs(int a)
```

Allocates no memory.

```
@SCJAllowed
@SCJMaySelfSuspend(false)
@SCJMayAllocate({})
@SCJPhase({
    javax.safetycritical.annotate.Phase.STARTUP,
    javax.safetycritical.annotate.Phase.INITIALIZATION,
    javax.safetycritical.annotate.Phase.RUN,
    javax.safetycritical.annotate.Phase.CLEANUP })
public static double acos(double a)
```

Allocates no memory.

```
@SCJAllowed
@SCJMaySelfSuspend(false)
@SCJMayAllocate({})
@SCJPhase({
    javax.safetycritical.annotate.Phase.STARTUP,
    javax.safetycritical.annotate.Phase.INITIALIZATION,
    javax.safetycritical.annotate.Phase.RUN,
    javax.safetycritical.annotate.Phase.CLEANUP })
public static double asin(double a)
```

Allocates no memory.

```
@SCJAllowed
@SCJMaySelfSuspend(false)
@SCJMayAllocate({})
@SCJPhase({
    javax.safetycritical.annotate.Phase.STARTUP,
    javax.safetycritical.annotate.Phase.INITIALIZATION,
    javax.safetycritical.annotate.Phase.RUN,
    javax.safetycritical.annotate.Phase.CLEANUP })
public static double atan(double a)
```

Allocates no memory.

```
@SCJAllowed
@SCJMaySelfSuspend(false)
@SCJMayAllocate({})
@SCJPhase({
    javax.safetycritical.annotate.Phase.STARTUP,
    javax.safetycritical.annotate.Phase.INITIALIZATION,
    javax.safetycritical.annotate.Phase.RUN,
    javax.safetycritical.annotate.Phase.CLEANUP })
public static double atan2(double a, double b)
```

Allocates no memory.

```
@SCJAllowed
@SCJMaySelfSuspend(false)
@SCJMayAllocate({})
@SCJPhase({
    javax.safetycritical.annotate.Phase.STARTUP,
    javax.safetycritical.annotate.Phase.INITIALIZATION,
    javax.safetycritical.annotate.Phase.RUN,
    javax.safetycritical.annotate.Phase.CLEANUP })
public static double cbrt(double a)
```

Allocates no memory.

```
@SCJAllowed
@SCJMaySelfSuspend(false)
@SCJMayAllocate({})
@SCJPhase({
    javax.safetycritical.annotate.Phase.STARTUP,
    javax.safetycritical.annotate.Phase.INITIALIZATION,
    javax.safetycritical.annotate.Phase.RUN,
    javax.safetycritical.annotate.Phase.CLEANUP })
public static double ceil(double a)
```

Allocates no memory.

```
@SCJAllowed
@SCJMaySelfSuspend(false)
@SCJMayAllocate({})
@SCJPhase({
    javax.safetycritical.annotate.Phase.STARTUP,
    javax.safetycritical.annotate.Phase.INITIALIZATION,
    javax.safetycritical.annotate.Phase.RUN,
    javax.safetycritical.annotate.Phase.CLEANUP })
public static double copySign(float magnitude, float sign)
```

Allocates no memory.

```
@SCJAllowed
@SCJMaySelfSuspend(false)
@SCJMayAllocate({})
@SCJPhase({
    javax.safetycritical.annotate.Phase.STARTUP,
    javax.safetycritical.annotate.Phase.INITIALIZATION,
    javax.safetycritical.annotate.Phase.RUN,
    javax.safetycritical.annotate.Phase.CLEANUP })
public static double copySign(double magnitude, double sign)
```

Allocates no memory.

```
@SCJAllowed
@SCJMaySelfSuspend(false)
@SCJMayAllocate({})
@SCJPhase({
    javax.safetycritical.annotate.Phase.STARTUP,
    javax.safetycritical.annotate.Phase.INITIALIZATION,
    javax.safetycritical.annotate.Phase.RUN,
    javax.safetycritical.annotate.Phase.CLEANUP })
public static double cos(double a)
```

Allocates no memory.

```
@SCJAllowed
@SCJMaySelfSuspend(false)
@SCJMayAllocate({})
@SCJPhase({
    javax.safetycritical.annotate.Phase.STARTUP,
    javax.safetycritical.annotate.Phase.INITIALIZATION,
    javax.safetycritical.annotate.Phase.RUN,
    javax.safetycritical.annotate.Phase.CLEANUP })
public static double cosh(double a)
```

Allocates no memory.

```
@SCJAllowed
@SCJMaySelfSuspend(false)
@SCJMayAllocate({})
@SCJPhase({
    javax.safetycritical.annotate.Phase.STARTUP,
    javax.safetycritical.annotate.Phase.INITIALIZATION,
    javax.safetycritical.annotate.Phase.RUN,
    javax.safetycritical.annotate.Phase.CLEANUP })
public static double exp(double a)
```

Allocates no memory.

```
@SCJAllowed
@SCJMaySelfSuspend(false)
@SCJMayAllocate({})
@SCJPhase({
    javax.safetycritical.annotate.Phase.STARTUP,
    javax.safetycritical.annotate.Phase.INITIALIZATION,
    javax.safetycritical.annotate.Phase.RUN,
    javax.safetycritical.annotate.Phase.CLEANUP })
public static double expm1(double a)
```

Allocates no memory.

```
@SCJAllowed
@SCJMaySelfSuspend(false)
@SCJMayAllocate({})
@SCJPhase({
    javax.safetycritical.annotate.Phase.STARTUP,
    javax.safetycritical.annotate.Phase.INITIALIZATION,
    javax.safetycritical.annotate.Phase.RUN,
    javax.safetycritical.annotate.Phase.CLEANUP })
public static double floor(double a)
```

Allocates no memory.

```
@SCJAllowed
@SCJMaySelfSuspend(false)
@SCJMayAllocate({})
@SCJPhase({
    javax.safetycritical.annotate.Phase.STARTUP,
    javax.safetycritical.annotate.Phase.INITIALIZATION,
    javax.safetycritical.annotate.Phase.RUN,
    javax.safetycritical.annotate.Phase.CLEANUP })
public static int getExponent(float a)
```

Allocates no memory.

```
@SCJAllowed
@SCJMaySelfSuspend(false)
@SCJMayAllocate({})
@SCJPhase({
    javax.safetycritical.annotate.Phase.STARTUP,
    javax.safetycritical.annotate.Phase.INITIALIZATION,
    javax.safetycritical.annotate.Phase.RUN,
    javax.safetycritical.annotate.Phase.CLEANUP })
public static int getExponent(double a)
```

Allocates no memory.

```
@SCJAllowed
@SCJMaySelfSuspend(false)
@SCJMayAllocate({})
@SCJPhase({
    javax.safetycritical.annotate.Phase.STARTUP,
    javax.safetycritical.annotate.Phase.INITIALIZATION,
    javax.safetycritical.annotate.Phase.RUN,
    javax.safetycritical.annotate.Phase.CLEANUP })
public static double hypot(double x, double y)
```

Allocates no memory.

```
@SCJAllowed
@SCJMaySelfSuspend(false)
@SCJMayAllocate({})
@SCJPhase({
    javax.safetycritical.annotate.Phase.STARTUP,
    javax.safetycritical.annotate.Phase.INITIALIZATION,
    javax.safetycritical.annotate.Phase.RUN,
    javax.safetycritical.annotate.Phase.CLEANUP })
public static double log(double a)
```

Allocates no memory.

```
@SCJAllowed
@SCJMaySelfSuspend(false)
@SCJMayAllocate({})
@SCJPhase({
    javax.safetycritical.annotate.Phase.STARTUP,
    javax.safetycritical.annotate.Phase.INITIALIZATION,
    javax.safetycritical.annotate.Phase.RUN,
    javax.safetycritical.annotate.Phase.CLEANUP })
public static double log10(double a)
```

Allocates no memory.

```
@SCJAllowed
@SCJMaySelfSuspend(false)
@SCJMayAllocate({})
@SCJPhase({
    javax.safetycritical.annotate.Phase.STARTUP,
    javax.safetycritical.annotate.Phase.INITIALIZATION,
    javax.safetycritical.annotate.Phase.RUN,
    javax.safetycritical.annotate.Phase.CLEANUP })
public static double log1p(double a)
```

Allocates no memory.

```
@SCJAllowed
@SCJMaySelfSuspend(false)
@SCJMayAllocate({})
@SCJPhase({
    javax.safetycritical.annotate.Phase.STARTUP,
    javax.safetycritical.annotate.Phase.INITIALIZATION,
    javax.safetycritical.annotate.Phase.RUN,
    javax.safetycritical.annotate.Phase.CLEANUP })
public static double max(double a, double b)
```

Allocates no memory.

```
@SCJAllowed
@SCJMaySelfSuspend(false)
@SCJMayAllocate({})
@SCJPhase({
    javax.safetycritical.annotate.Phase.STARTUP,
    javax.safetycritical.annotate.Phase.INITIALIZATION,
    javax.safetycritical.annotate.Phase.RUN,
    javax.safetycritical.annotate.Phase.CLEANUP })
public static long max(long a, long b)
```

Allocates no memory.

```
@SCJAllowed
@SCJMaySelfSuspend(false)
@SCJMayAllocate({})
@SCJPhase({
    javax.safetycritical.annotate.Phase.STARTUP,
    javax.safetycritical.annotate.Phase.INITIALIZATION,
    javax.safetycritical.annotate.Phase.RUN,
    javax.safetycritical.annotate.Phase.CLEANUP })
public static float max(float a, float b)
```

Allocates no memory.

```
@SCJAllowed
@SCJMaySelfSuspend(false)
@SCJMayAllocate({})
@SCJPhase({
    javax.safetycritical.annotate.Phase.STARTUP,
    javax.safetycritical.annotate.Phase.INITIALIZATION,
    javax.safetycritical.annotate.Phase.RUN,
    javax.safetycritical.annotate.Phase.CLEANUP })
public static int max(int a, int b)
```

Allocates no memory.

```
@SCJAllowed
@SCJMaySelfSuspend(false)
@SCJMayAllocate({})
@SCJPhase({
    javax.safetycritical.annotate.Phase.STARTUP,
    javax.safetycritical.annotate.Phase.INITIALIZATION,
    javax.safetycritical.annotate.Phase.RUN,
    javax.safetycritical.annotate.Phase.CLEANUP })
public static long min(long a, long b)
```

Allocates no memory.

```
@SCJAllowed
@SCJMaySelfSuspend(false)
@SCJMayAllocate({})
@SCJPhase({
    javax.safetycritical.annotate.Phase.STARTUP,
    javax.safetycritical.annotate.Phase.INITIALIZATION,
    javax.safetycritical.annotate.Phase.RUN,
    javax.safetycritical.annotate.Phase.CLEANUP })
public static double min(double a, double b)
```

Allocates no memory.

```
@SCJAllowed
@SCJMaySelfSuspend(false)
@SCJMayAllocate({})
@SCJPhase({
    javax.safetycritical.annotate.Phase.STARTUP,
    javax.safetycritical.annotate.Phase.INITIALIZATION,
    javax.safetycritical.annotate.Phase.RUN,
    javax.safetycritical.annotate.Phase.CLEANUP })
public static float min(float a, float b)
```

Allocates no memory.

```
@SCJAllowed
@SCJMaySelfSuspend(false)
@SCJMayAllocate({})
@SCJPhase({
    javax.safetycritical.annotate.Phase.STARTUP,
    javax.safetycritical.annotate.Phase.INITIALIZATION,
    javax.safetycritical.annotate.Phase.RUN,
    javax.safetycritical.annotate.Phase.CLEANUP })
public static int min(int a, int b)
```

Allocates no memory.

```
@SCJAllowed
@SCJMaySelfSuspend(false)
@SCJMayAllocate({})
@SCJPhase({
    javax.safetycritical.annotate.Phase.STARTUP,
    javax.safetycritical.annotate.Phase.INITIALIZATION,
    javax.safetycritical.annotate.Phase.RUN,
    javax.safetycritical.annotate.Phase.CLEANUP })
public static float nextAfter(float start, float direction)
```

Allocates no memory.

```
@SCJAllowed
@SCJMaySelfSuspend(false)
@SCJMayAllocate({})
@SCJPhase({
    javax.safetycritical.annotate.Phase.STARTUP,
    javax.safetycritical.annotate.Phase.INITIALIZATION,
    javax.safetycritical.annotate.Phase.RUN,
    javax.safetycritical.annotate.Phase.CLEANUP })
public static double nextAfter(double start, double direction)
```

Allocates no memory.

```
@SCJAllowed
@SCJMaySelfSuspend(false)
@SCJMayAllocate({})
@SCJPhase({
    javax.safetycritical.annotate.Phase.STARTUP,
    javax.safetycritical.annotate.Phase.INITIALIZATION,
    javax.safetycritical.annotate.Phase.RUN,
    javax.safetycritical.annotate.Phase.CLEANUP })
public static float nextUp(float d)
```

Allocates no memory.

```
@SCJAllowed
@SCJMaySelfSuspend(false)
@SCJMayAllocate({})
@SCJPhase({
    javax.safetycritical.annotate.Phase.STARTUP,
    javax.safetycritical.annotate.Phase.INITIALIZATION,
    javax.safetycritical.annotate.Phase.RUN,
    javax.safetycritical.annotate.Phase.CLEANUP })
public static double nextUp(double d)
```

Allocates no memory.

```
@SCJAllowed
@SCJMaySelfSuspend(false)
@SCJMayAllocate({})
@SCJPhase({
    javax.safetycritical.annotate.Phase.STARTUP,
    javax.safetycritical.annotate.Phase.INITIALIZATION,
    javax.safetycritical.annotate.Phase.RUN,
    javax.safetycritical.annotate.Phase.CLEANUP })
public static double pow(double a, double b)
```

Allocates no memory.

```
@SCJAllowed
@SCJMaySelfSuspend(false)
@SCJMayAllocate({})
@SCJPhase({
    javax.safetycritical.annotate.Phase.STARTUP,
    javax.safetycritical.annotate.Phase.INITIALIZATION,
    javax.safetycritical.annotate.Phase.RUN,
    javax.safetycritical.annotate.Phase.CLEANUP })
public static double random( )
```

Allocates no memory.

```
@SCJAllowed
@SCJMaySelfSuspend(false)
@SCJMayAllocate({})
@SCJPhase({
    javax.safetycritical.annotate.Phase.STARTUP,
    javax.safetycritical.annotate.Phase.INITIALIZATION,
    javax.safetycritical.annotate.Phase.RUN,
    javax.safetycritical.annotate.Phase.CLEANUP })
public static double rint(double a)
```

Allocates no memory.

```
@SCJAllowed
@SCJMaySelfSuspend(false)
@SCJMayAllocate({})
@SCJPhase({
    javax.safetycritical.annotate.Phase.STARTUP,
    javax.safetycritical.annotate.Phase.INITIALIZATION,
    javax.safetycritical.annotate.Phase.RUN,
    javax.safetycritical.annotate.Phase.CLEANUP })
public static long round(double a)
```

Allocates no memory.

```
@SCJAllowed
@SCJMaySelfSuspend(false)
@SCJMayAllocate({})
@SCJPhase({
    javax.safetycritical.annotate.Phase.STARTUP,
    javax.safetycritical.annotate.Phase.INITIALIZATION,
    javax.safetycritical.annotate.Phase.RUN,
    javax.safetycritical.annotate.Phase.CLEANUP })
public static int round(float a)
```

Allocates no memory.

```
@SCJAllowed
@SCJMaySelfSuspend(false)
@SCJMayAllocate({})
@SCJPhase({
    javax.safetycritical.annotate.Phase.STARTUP,
    javax.safetycritical.annotate.Phase.INITIALIZATION,
    javax.safetycritical.annotate.Phase.RUN,
    javax.safetycritical.annotate.Phase.CLEANUP })
public static float scalb(float f, int scaleFactor)
```

Allocates no memory.

```
@SCJAllowed
@SCJMaySelfSuspend(false)
@SCJMayAllocate({})
@SCJPhase({
    javax.safetycritical.annotate.Phase.STARTUP,
    javax.safetycritical.annotate.Phase.INITIALIZATION,
    javax.safetycritical.annotate.Phase.RUN,
    javax.safetycritical.annotate.Phase.CLEANUP })
public static double scalb(double d, int scaleFactor)
```

Allocates no memory.

```
@SCJAllowed
@SCJMaySelfSuspend(false)
@SCJMayAllocate({})
@SCJPhase({
    javax.safetycritical.annotate.Phase.STARTUP,
    javax.safetycritical.annotate.Phase.INITIALIZATION,
    javax.safetycritical.annotate.Phase.RUN,
    javax.safetycritical.annotate.Phase.CLEANUP })
public static float signum(float f)
```

Allocates no memory.

```
@SCJAllowed
@SCJMaySelfSuspend(false)
@SCJMayAllocate({})
@SCJPhase({
    javax.safetycritical.annotate.Phase.STARTUP,
    javax.safetycritical.annotate.Phase.INITIALIZATION,
    javax.safetycritical.annotate.Phase.RUN,
    javax.safetycritical.annotate.Phase.CLEANUP })
public static double signum(double d)
```

Allocates no memory.

```
@SCJAllowed
@SCJMaySelfSuspend(false)
@SCJMayAllocate({})
@SCJPhase({
    javax.safetycritical.annotate.Phase.STARTUP,
    javax.safetycritical.annotate.Phase.INITIALIZATION,
    javax.safetycritical.annotate.Phase.RUN,
    javax.safetycritical.annotate.Phase.CLEANUP })
public static double sin(double a)
```

Allocates no memory.

```
@SCJAllowed
@SCJMaySelfSuspend(false)
@SCJMayAllocate({})
@SCJPhase({
    javax.safetycritical.annotate.Phase.STARTUP,
    javax.safetycritical.annotate.Phase.INITIALIZATION,
    javax.safetycritical.annotate.Phase.RUN,
    javax.safetycritical.annotate.Phase.CLEANUP })
public static double sinh(double a)
```

Allocates no memory.

```
@SCJAllowed
@SCJMaySelfSuspend(false)
@SCJMayAllocate({})
@SCJPhase({
    javax.safetycritical.annotate.Phase.STARTUP,
    javax.safetycritical.annotate.Phase.INITIALIZATION,
    javax.safetycritical.annotate.Phase.RUN,
    javax.safetycritical.annotate.Phase.CLEANUP })
public static double sqrt(double a)
```

Allocates no memory.

```
@SCJAllowed
@SCJMaySelfSuspend(false)
@SCJMayAllocate({})
@SCJPhase({
    javax.safetycritical.annotate.Phase.STARTUP,
    javax.safetycritical.annotate.Phase.INITIALIZATION,
    javax.safetycritical.annotate.Phase.RUN,
    javax.safetycritical.annotate.Phase.CLEANUP })
public static double tan(double a)
```

Allocates no memory.

```
@SCJAllowed
@SCJMaySelfSuspend(false)
@SCJMayAllocate({})
@SCJPhase({
    javax.safetycritical.annotate.Phase.STARTUP,
    javax.safetycritical.annotate.Phase.INITIALIZATION,
    javax.safetycritical.annotate.Phase.RUN,
    javax.safetycritical.annotate.Phase.CLEANUP })
public static double tanh(double a)
```

Allocates no memory.

```
@SCJAllowed
@SCJMaySelfSuspend(false)
@SCJMayAllocate({})
@SCJPhase({
    javax.safetycritical.annotate.Phase.STARTUP,
    javax.safetycritical.annotate.Phase.INITIALIZATION,
    javax.safetycritical.annotate.Phase.RUN,
    javax.safetycritical.annotate.Phase.CLEANUP })
public static double toDegrees(double val)
```

Allocates no memory.

```
@SCJAllowed
@SCJMaySelfSuspend(false)
@SCJMayAllocate({})
@SCJPhase({
    javax.safetycritical.annotate.Phase.STARTUP,
    javax.safetycritical.annotate.Phase.INITIALIZATION,
    javax.safetycritical.annotate.Phase.RUN,
    javax.safetycritical.annotate.Phase.CLEANUP })
public static double toRadians(double val)
```

Allocates no memory.

```
@SCJAllowed
@SCJMaySelfSuspend(false)
@SCJMayAllocate({})
@SCJPhase({
    javax.safetycritical.annotate.Phase.STARTUP,
    javax.safetycritical.annotate.Phase.INITIALIZATION,
    javax.safetycritical.annotate.Phase.RUN,
    javax.safetycritical.annotate.Phase.CLEANUP })
public static float ulp(float d)
```

Allocates no memory.

```
@SCJAllowed
@SCJMaySelfSuspend(false)
@SCJMayAllocate({})
@SCJPhase({
    javax.safetycritical.annotate.Phase.STARTUP,
    javax.safetycritical.annotate.Phase.INITIALIZATION,
    javax.safetycritical.annotate.Phase.RUN,
    javax.safetycritical.annotate.Phase.CLEANUP })
public static double ulp(double d)
```

Allocates no memory.

B.3.29 CLASS NegativeArraySizeException

```
@SCJAllowed
public class NegativeArraySizeException implements java.io.Serializable
      extends java.lang.RuntimeException
```

Thrown if an application tries to create an array with negative size.

Constructors

```
@SCJAllowed
@SCJMaySelfSuspend(false)
@SCJMayAllocate({})
@SCJPhase({
    javax.safetycritical.annotate.Phase.STARTUP,
    javax.safetycritical.annotate.Phase.INITIALIZATION,
    javax.safetycritical.annotate.Phase.RUN,
    javax.safetycritical.annotate.Phase.CLEANUP })
public NegativeArraySizeException( )
```

Shall not copy ”this” to any instance or static field.

Allocates an application- and implementation-dependent amount of memory in the current scope (to represent stack backtrace).

```
@SCJAllowed
@SCJMaySelfSuspend(false)
@SCJMayAllocate({})
@SCJPhase({
   javax.safetycritical.annotate.Phase.STARTUP,
   javax.safetycritical.annotate.Phase.INITIALIZATION,
   javax.safetycritical.annotate.Phase.RUN,
   javax.safetycritical.annotate.Phase.CLEANUP })
public NegativeArraySizeException(String msg)
```

Shall not copy ”this” to any instance or static field. The scope containing the msg argument must enclose the scope containing ”this”. Otherwise, an IllegalAssignmentError will be thrown.

Allocates an application- and implementation-dependent amount of memory in the current scope (to represent stack backtrace).

B.3.30 CLASS NullPointerException

```
@SCJAllowed
public class NullPointerException implements java.io.Serializable extends
    java.lang.RuntimeException
```

Thrown when an application attempts to use null in a case where an object is required.

Constructors

```
@SCJAllowed
@SCJMaySelfSuspend(false)
@SCJMayAllocate({})
@SCJPhase({
   javax.safetycritical.annotate.Phase.STARTUP,
   javax.safetycritical.annotate.Phase.INITIALIZATION,
   javax.safetycritical.annotate.Phase.RUN,
   javax.safetycritical.annotate.Phase.CLEANUP })
public NullPointerException( )
```

Shall not copy ”this” to any instance or static field.

Allocates an application- and implementation-dependent amount of memory in the current scope (to represent stack backtrace).

```
@SCJAllowed
@SCJMaySelfSuspend(false)
@SCJMayAllocate({})
@SCJPhase({
   javax.safetycritical.annotate.Phase.STARTUP,
   javax.safetycritical.annotate.Phase.INITIALIZATION,
   javax.safetycritical.annotate.Phase.RUN,
   javax.safetycritical.annotate.Phase.CLEANUP })
public NullPointerException(String msg)
```

Shall not copy ”this” to any instance or static field. The scope containing the msg argument must enclose the scope containing ”this”. Otherwise, an IllegalAssignmentError will be thrown.

Allocates an application- and implementation-dependent amount of memory in the current scope (to represent stack backtrace).

B.3.31 CLASS **Number**

```
@SCJAllowed
public abstract class Number implements java.io.Serializable extends
    java.lang.Object
```

The abstract class Number is the superclass of classes BigDecimal, BigInteger, Byte, Double, Float, Integer, Long, and Short.

Constructors

```
@SCJAllowed
@SCJMaySelfSuspend(false)
@SCJMayAllocate({})
@SCJPhase({
   javax.safetycritical.annotate.Phase.STARTUP,
   javax.safetycritical.annotate.Phase.INITIALIZATION,
   javax.safetycritical.annotate.Phase.RUN,
   javax.safetycritical.annotate.Phase.CLEANUP })
public Number( )
```

The implementation of this method shall not allow ”this” to escape the method’s local variables.

Methods

```
@SCJAllowed
@SCJMaySelfSuspend(false)
@SCJMayAllocate({})
@SCJPhase({
   javax.safetycritical.annotate.Phase.STARTUP,
   javax.safetycritical.annotate.Phase.INITIALIZATION,
   javax.safetycritical.annotate.Phase.RUN,
   javax.safetycritical.annotate.Phase.CLEANUP })
public byte byteValue( )
```

The implementation of this method shall not allow ”this” to escape the method’s local variables.

```
@SCJAllowed
@SCJMaySelfSuspend(false)
@SCJMayAllocate({})
@SCJPhase({
   javax.safetycritical.annotate.Phase.STARTUP,
   javax.safetycritical.annotate.Phase.INITIALIZATION,
   javax.safetycritical.annotate.Phase.RUN,
   javax.safetycritical.annotate.Phase.CLEANUP })
public abstract double doubleValue( )
```

The implementation of this method shall not allow ”this” to escape the method’s local variables.

```
@SCJAllowed
@SCJMaySelfSuspend(false)
@SCJMayAllocate({})
@SCJPhase({
   javax.safetycritical.annotate.Phase.STARTUP,
   javax.safetycritical.annotate.Phase.INITIALIZATION,
   javax.safetycritical.annotate.Phase.RUN,
   javax.safetycritical.annotate.Phase.CLEANUP })
public abstract float floatValue( )
```

The implementation of this method shall not allow ”this” to escape the method’s local variables.

```
@SCJAllowed
@SCJMaySelfSuspend(false)
@SCJMayAllocate({})
@SCJPhase({
   javax.safetycritical.annotate.Phase.STARTUP,
   javax.safetycritical.annotate.Phase.INITIALIZATION,
   javax.safetycritical.annotate.Phase.RUN,
   javax.safetycritical.annotate.Phase.CLEANUP })
public abstract int intValue( )
```

The implementation of this method shall not allow ”this” to escape the method’s local variables.

```
@SCJAllowed
@SCJMaySelfSuspend(false)
@SCJMayAllocate({})
@SCJPhase({
    javax.safetycritical.annotate.Phase.STARTUP,
    javax.safetycritical.annotate.Phase.INITIALIZATION,
    javax.safetycritical.annotate.Phase.RUN,
    javax.safetycritical.annotate.Phase.CLEANUP })
public abstract long longValue( )
```

The implementation of this method shall not allow ”this” to escape the method’s local variables.

```
@SCJAllowed
@SCJMaySelfSuspend(false)
@SCJMayAllocate({})
@SCJPhase({
    javax.safetycritical.annotate.Phase.STARTUP,
    javax.safetycritical.annotate.Phase.INITIALIZATION,
    javax.safetycritical.annotate.Phase.RUN,
    javax.safetycritical.annotate.Phase.CLEANUP })
public abstract short shortValue( )
```

The implementation of this method shall not allow ”this” to escape the method’s local variables.

B.3.32 CLASS NumberFormatException

```
@SCJAllowed
public class NumberFormatException implements java.io.Serializable extends
    java.lang.IllegalArgumentException
```

Thrown to indicate that the application has attempted to convert a string to one of the numeric types, but that the string does not have the appropriate format.

Constructors

```
@SCJAllowed
@SCJMaySelfSuspend(false)
@SCJMayAllocate({})
@SCJPhase({
```

```
    javax.safetycritical.annotate.Phase.STARTUP,
    javax.safetycritical.annotate.Phase.INITIALIZATION,
    javax.safetycritical.annotate.Phase.RUN,
    javax.safetycritical.annotate.Phase.CLEANUP })
public NumberFormatException( )
```

Shall not copy ”this” to any instance or static field.

Allocates an application- and implementation-dependent amount of memory in the current scope (to represent stack backtrace).

```
@SCJAllowed
@SCJMaySelfSuspend(false)
@SCJMayAllocate({})
@SCJPhase({
    javax.safetycritical.annotate.Phase.STARTUP,
    javax.safetycritical.annotate.Phase.INITIALIZATION,
    javax.safetycritical.annotate.Phase.RUN,
    javax.safetycritical.annotate.Phase.CLEANUP })
public NumberFormatException(String msg)
```

Shall not copy ”this” to any instance or static field. The scope containing the msg argument must enclose the scope containing ”this”. Otherwise, an IllegalAssignmentError will be thrown.

Allocates an application- and implementation-dependent amount of memory in the current scope (to represent stack backtrace).

B.3.33 CLASS Object

```
@SCJAllowed
public class Object
```

Class Object is the root of the class hierarchy. Every class has Object as a superclass. All objects, including arrays, implement the methods of this class.

Constructors

```
@SCJAllowed
@SCJMaySelfSuspend(false)
@SCJMayAllocate({})
@SCJPhase({
    javax.safetycritical.annotate.Phase.STARTUP,
    javax.safetycritical.annotate.Phase.INITIALIZATION,
    javax.safetycritical.annotate.Phase.RUN,
    javax.safetycritical.annotate.Phase.CLEANUP })
public Object( )
```

Allocates no memory. Does not allow ”this” to escape local variables.

Methods

```
@SCJAllowed
@SCJMaySelfSuspend(false)
@SCJMayAllocate({})
@SCJPhase({
   javax.safetycritical.annotate.Phase.STARTUP,
   javax.safetycritical.annotate.Phase.INITIALIZATION,
   javax.safetycritical.annotate.Phase.RUN,
   javax.safetycritical.annotate.Phase.CLEANUP })
public boolean equals(Object obj)
```

Allocates no memory. Does not allow ”this” or ”obj” argument to escape local variables.

```
@SCJAllowed
@SCJMaySelfSuspend(false)
@SCJMayAllocate({})
@SCJPhase({
   javax.safetycritical.annotate.Phase.STARTUP,
   javax.safetycritical.annotate.Phase.INITIALIZATION,
   javax.safetycritical.annotate.Phase.RUN,
   javax.safetycritical.annotate.Phase.CLEANUP })
public final java.lang.Class<? extends java.lang.Object> getClass( )
```

Allocates no memory. Does not allow ”this” or ”obj” argument to escape local variables.

```
@SCJAllowed
@SCJMaySelfSuspend(false)
@SCJMayAllocate({})
@SCJPhase({
   javax.safetycritical.annotate.Phase.STARTUP,
   javax.safetycritical.annotate.Phase.INITIALIZATION,
   javax.safetycritical.annotate.Phase.RUN,
   javax.safetycritical.annotate.Phase.CLEANUP })
public int hashCode( )
```

Allocates no memory. Does not allow ”this” to escape local variables.

```
@SCJAllowed(javax.safetycritical.annotate.Level.LEVEL_2)
@SCJMaySelfSuspend(false)
@SCJMayAllocate({})
@SCJPhase({
   javax.safetycritical.annotate.Phase.STARTUP,
   javax.safetycritical.annotate.Phase.INITIALIZATION,
   javax.safetycritical.annotate.Phase.RUN,
   javax.safetycritical.annotate.Phase.CLEANUP })
public final void notify( )
```

Allocates no memory. Does not allow ”this” to escape local variables.

```
@SCJAllowed(javax.safetycritical.annotate.Level.LEVEL_2)
@SCJMaySelfSuspend(false)
@SCJMayAllocate({})
@SCJPhase({
   javax.safetycritical.annotate.Phase.STARTUP,
   javax.safetycritical.annotate.Phase.INITIALIZATION,
   javax.safetycritical.annotate.Phase.RUN,
   javax.safetycritical.annotate.Phase.CLEANUP })
public final void notifyAll( )
```

Allocates no memory. Does not allow ”this” to escape local variables.

```
@SCJAllowed
@SCJMaySelfSuspend(false)
@SCJMayAllocate({javax.safetycritical.annotate.AllocationContext.CURRENT})
@SCJPhase({
   javax.safetycritical.annotate.Phase.STARTUP,
   javax.safetycritical.annotate.Phase.INITIALIZATION,
   javax.safetycritical.annotate.Phase.RUN,
   javax.safetycritical.annotate.Phase.CLEANUP })
public java.lang.String toString( )
```

Does not allow ”this” to escape local variables. Allocates a String and associated internal ”structure” (e.g. char[]) in caller’s scope.

```
@SCJAllowed(javax.safetycritical.annotate.Level.LEVEL_2)
@SCJMaySelfSuspend(true)
@SCJMayAllocate({})
@SCJPhase({
   javax.safetycritical.annotate.Phase.STARTUP,
   javax.safetycritical.annotate.Phase.INITIALIZATION,
   javax.safetycritical.annotate.Phase.RUN,
   javax.safetycritical.annotate.Phase.CLEANUP })
public final void wait(long timeout, int nanos)
  throws java.lang.InterruptedException
```

Allocates no memory. Does not allow ”this” to escape local variables.

Throws InterruptedException

```
@SCJAllowed(javax.safetycritical.annotate.Level.LEVEL_2)
@SCJMaySelfSuspend(true)
@SCJMayAllocate({})
@SCJPhase({
    javax.safetycritical.annotate.Phase.STARTUP,
    javax.safetycritical.annotate.Phase.INITIALIZATION,
    javax.safetycritical.annotate.Phase.RUN,
    javax.safetycritical.annotate.Phase.CLEANUP })
public final void wait(long timeout)
  throws java.lang.InterruptedException
```

Allocates no memory. Does not allow ”this” to escape local variables.

Throws InterruptedException

```
@SCJAllowed(javax.safetycritical.annotate.Level.LEVEL_2)
@SCJMaySelfSuspend(true)
@SCJMayAllocate({})
@SCJPhase({
    javax.safetycritical.annotate.Phase.STARTUP,
    javax.safetycritical.annotate.Phase.INITIALIZATION,
    javax.safetycritical.annotate.Phase.RUN,
    javax.safetycritical.annotate.Phase.CLEANUP })
public final void wait( )
```

Allocates no memory. Does not allow ”this” to escape local variables.

Throws InterruptedException

B.3.34 CLASS OutOfMemoryError

```
@SCJAllowed
public class OutOfMemoryError implements java.io.Serializable extends
    java.lang.VirtualMachineError
```

Thrown when the Java Virtual Machine cannot allocate an object because it is out of memory.

Constructors

```
@SCJAllowed
@SCJMaySelfSuspend(false)
@SCJMayAllocate({})
@SCJPhase({
   javax.safetycritical.annotate.Phase.STARTUP,
   javax.safetycritical.annotate.Phase.INITIALIZATION,
   javax.safetycritical.annotate.Phase.RUN,
   javax.safetycritical.annotate.Phase.CLEANUP })
public OutOfMemoryError( )
```

Shall not copy ”this” to any instance or static field.

Allocates an application- and implementation-dependent amount of memory in the current scope (to represent stack backtrace).

```
@SCJAllowed
@SCJMaySelfSuspend(false)
@SCJMayAllocate({})
@SCJPhase({
   javax.safetycritical.annotate.Phase.STARTUP,
   javax.safetycritical.annotate.Phase.INITIALIZATION,
   javax.safetycritical.annotate.Phase.RUN,
   javax.safetycritical.annotate.Phase.CLEANUP })
public OutOfMemoryError(String msg)
```

Shall not copy ”this” to any instance or static field. The scope containing the msg argument must enclose the scope containing ”this”. Otherwise, an IllegalAssignmentError will be thrown.

Allocates an application- and implementation-dependent amount of memory in the current scope (to represent stack backtrace).

B.3.35 CLASS **RuntimeException**

```
@SCJAllowed
public class RuntimeException implements java.io.Serializable extends
    java.lang.Exception
```

RuntimeException is the superclass of those exceptions that can be thrown during the normal operation of the Java Virtual Machine.

RuntimeException and its subclasses are unchecked exceptions. Unchecked exceptions do not need to be declared in a method or constructor’s throws clause if they can be thrown by the execution of the method or constructor and propagate outside the method or constructor boundary.

Constructors

```
@SCJAllowed
@SCJMaySelfSuspend(false)
@SCJMayAllocate({})
@SCJPhase({
   javax.safetycritical.annotate.Phase.STARTUP,
   javax.safetycritical.annotate.Phase.INITIALIZATION,
   javax.safetycritical.annotate.Phase.RUN,
   javax.safetycritical.annotate.Phase.CLEANUP })
public RuntimeException( )
```

Shall not copy ”this” to any instance or static field.

Allocates an application- and implementation-dependent amount of memory in the current scope (to represent stack backtrace).

```
@SCJAllowed
@SCJMaySelfSuspend(false)
@SCJMayAllocate({})
@SCJPhase({
   javax.safetycritical.annotate.Phase.STARTUP,
   javax.safetycritical.annotate.Phase.INITIALIZATION,
   javax.safetycritical.annotate.Phase.RUN,
   javax.safetycritical.annotate.Phase.CLEANUP })
public RuntimeException(String msg)
```

Shall not copy ”this” to any instance or static field. The scope containing the msg argument must enclose the scope containing ”this”. Otherwise, an IllegalAssignmentError will be thrown.

Allocates an application- and implementation-dependent amount of memory in the current scope (to represent stack backtrace).

```
@SCJAllowed
@SCJMaySelfSuspend(false)
@SCJMayAllocate({})
@SCJPhase({
   javax.safetycritical.annotate.Phase.STARTUP,
   javax.safetycritical.annotate.Phase.INITIALIZATION,
   javax.safetycritical.annotate.Phase.RUN,
   javax.safetycritical.annotate.Phase.CLEANUP })
public RuntimeException(String msg, Throwable t)
```

Shall not copy ”this” to any instance or static field. The scope containing the msg argument must enclose the scope containing ”this”. Otherwise, an IllegalAssignmentError will be thrown.

Allocates an application- and implementation-dependent amount of memory in the current scope (to represent stack backtrace).

```
@SCJAllowed
@SCJMaySelfSuspend(false)
@SCJMayAllocate({})
@SCJPhase({
   javax.safetycritical.annotate.Phase.STARTUP,
   javax.safetycritical.annotate.Phase.INITIALIZATION,
   javax.safetycritical.annotate.Phase.RUN,
   javax.safetycritical.annotate.Phase.CLEANUP })
public RuntimeException(Throwable t)
```

Shall not copy ”this” to any instance or static field. The scope containing the msg argument must enclose the scope containing ”this”. Otherwise, an IllegalAssignmentError will be thrown.

Allocates an application- and implementation-dependent amount of memory in the current scope (to represent stack backtrace).

B.3.36 CLASS **Short**

```
@SCJAllowed
public class Short implements java.lang.Comparable<T>, java.io.Serializable
    extends java.lang.Number
```

The Short class wraps a value of primitive type short in an object. An object of type Short contains a single field whose type is short.

Fields

```
@SCJAllowed
public static final short MAX_VALUE
```

```
@SCJAllowed
public static final short MIN_VALUE
```

```
@SCJAllowed
public static final int SIZE
```

```
@SCJAllowed
public static final java.lang.Class<java.lang.Short> TYPE
```

Constructors

```
@SCJAllowed
@SCJMaySelfSuspend(false)
@SCJMayAllocate({
    javax.safetycritical.annotate.AllocationContext.CURRENT,
    javax.safetycritical.annotate.AllocationContext.INNER,
    javax.safetycritical.annotate.AllocationContext.OUTER})
@SCJPhase({
    javax.safetycritical.annotate.Phase.STARTUP,
    javax.safetycritical.annotate.Phase.INITIALIZATION,
    javax.safetycritical.annotate.Phase.RUN,
    javax.safetycritical.annotate.Phase.CLEANUP })
public Short(short val)
```

Allocates no memory. Does not allow ”this” to escape local variables.

```
@SCJAllowed
@SCJMaySelfSuspend(false)
@SCJMayAllocate({
    javax.safetycritical.annotate.AllocationContext.CURRENT,
    javax.safetycritical.annotate.AllocationContext.INNER,
    javax.safetycritical.annotate.AllocationContext.OUTER})
@SCJPhase({
    javax.safetycritical.annotate.Phase.STARTUP,
    javax.safetycritical.annotate.Phase.INITIALIZATION,
    javax.safetycritical.annotate.Phase.RUN,
    javax.safetycritical.annotate.Phase.CLEANUP })
public Short(String str)
  throws java.lang.NumberFormatException
```

Allocates no memory. Does not allow ”this” or ”str” argument to escape local variables.

Throws NumberFormatException

Methods

```
@SCJAllowed
@SCJMaySelfSuspend(false)
@SCJMayAllocate({})
@SCJPhase({
    javax.safetycritical.annotate.Phase.STARTUP,
    javax.safetycritical.annotate.Phase.INITIALIZATION,
    javax.safetycritical.annotate.Phase.RUN,
    javax.safetycritical.annotate.Phase.CLEANUP })
public byte byteValue( )
```

Allocates no memory. Does not allow ”this” to escape local variables.

```
@SCJAllowed
@SCJMaySelfSuspend(false)
@SCJMayAllocate({})
@SCJPhase({
   javax.safetycritical.annotate.Phase.STARTUP,
   javax.safetycritical.annotate.Phase.INITIALIZATION,
   javax.safetycritical.annotate.Phase.RUN,
   javax.safetycritical.annotate.Phase.CLEANUP })
public int compareTo(Short other)
```

Allocates no memory. Does not allow ”this” or ”other” argument to escape local variables.

```
@SCJAllowed
@SCJMaySelfSuspend(false)
@SCJMayAllocate({
   javax.safetycritical.annotate.AllocationContext.CURRENT,
   javax.safetycritical.annotate.AllocationContext.INNER,
   javax.safetycritical.annotate.AllocationContext.OUTER})
@SCJPhase({
   javax.safetycritical.annotate.Phase.STARTUP,
   javax.safetycritical.annotate.Phase.INITIALIZATION,
   javax.safetycritical.annotate.Phase.RUN,
   javax.safetycritical.annotate.Phase.CLEANUP })
public static java.lang.Short decode(String str)
  throws java.lang.NumberFormatException
```

Does not allow ”str” argument to escape local variables. Allocates a Short in caller’s scope.

Throws NumberFormatException

```
@SCJAllowed
@SCJMaySelfSuspend(false)
@SCJMayAllocate({})
@SCJPhase({
   javax.safetycritical.annotate.Phase.STARTUP,
   javax.safetycritical.annotate.Phase.INITIALIZATION,
   javax.safetycritical.annotate.Phase.RUN,
   javax.safetycritical.annotate.Phase.CLEANUP })
public double doubleValue( )
```

Allocates no memory.

```
@SCJAllowed
@SCJMaySelfSuspend(false)
@SCJMayAllocate({})
@SCJPhase({
    javax.safetycritical.annotate.Phase.STARTUP,
    javax.safetycritical.annotate.Phase.INITIALIZATION,
    javax.safetycritical.annotate.Phase.RUN,
    javax.safetycritical.annotate.Phase.CLEANUP })
public boolean equals(Object obj)
```

Allocates no memory. Does not allow ”this” or ”obj” argument to escape local variables.

```
@SCJAllowed
@SCJMaySelfSuspend(false)
@SCJMayAllocate({
    javax.safetycritical.annotate.AllocationContext.CURRENT,
    javax.safetycritical.annotate.AllocationContext.INNER,
    javax.safetycritical.annotate.AllocationContext.OUTER})
@SCJPhase({
    javax.safetycritical.annotate.Phase.STARTUP,
    javax.safetycritical.annotate.Phase.INITIALIZATION,
    javax.safetycritical.annotate.Phase.RUN,
    javax.safetycritical.annotate.Phase.CLEANUP })
public float floatValue( )
```

Allocates no memory. Does not allow ”this” to escape local variables.

```
@SCJAllowed
@SCJMaySelfSuspend(false)
@SCJMayAllocate({
    javax.safetycritical.annotate.AllocationContext.CURRENT,
    javax.safetycritical.annotate.AllocationContext.INNER,
    javax.safetycritical.annotate.AllocationContext.OUTER})
@SCJPhase({
    javax.safetycritical.annotate.Phase.STARTUP,
    javax.safetycritical.annotate.Phase.INITIALIZATION,
    javax.safetycritical.annotate.Phase.RUN,
    javax.safetycritical.annotate.Phase.CLEANUP })
public int hashCode( )
```

Allocates no memory. Does not allow ”this” to escape local variables.

```
@SCJAllowed
@SCJMaySelfSuspend(false)
@SCJMayAllocate({})
```

```
@SCJPhase({
    javax.safetycritical.annotate.Phase.STARTUP,
    javax.safetycritical.annotate.Phase.INITIALIZATION,
    javax.safetycritical.annotate.Phase.RUN,
    javax.safetycritical.annotate.Phase.CLEANUP })
public int intValue( )
```

Allocates no memory. Does not allow ”this” to escape local variables.

```
@SCJAllowed
@SCJMaySelfSuspend(false)
@SCJMayAllocate({})
@SCJPhase({
    javax.safetycritical.annotate.Phase.STARTUP,
    javax.safetycritical.annotate.Phase.INITIALIZATION,
    javax.safetycritical.annotate.Phase.RUN,
    javax.safetycritical.annotate.Phase.CLEANUP })
public long longValue( )
```

Allocates no memory. Does not allow ”this” to escape local variables.

```
@SCJAllowed
@SCJMaySelfSuspend(false)
@SCJMayAllocate({})
@SCJPhase({
    javax.safetycritical.annotate.Phase.STARTUP,
    javax.safetycritical.annotate.Phase.INITIALIZATION,
    javax.safetycritical.annotate.Phase.RUN,
    javax.safetycritical.annotate.Phase.CLEANUP })
public static short parseShort(String str, int base)
  throws java.lang.NumberFormatException
```

Allocates no memory. Does not allow ”str” argument to escape local variables.

```
@SCJAllowed
@SCJMaySelfSuspend(false)
@SCJMayAllocate({})
@SCJPhase({
    javax.safetycritical.annotate.Phase.STARTUP,
    javax.safetycritical.annotate.Phase.INITIALIZATION,
    javax.safetycritical.annotate.Phase.RUN,
    javax.safetycritical.annotate.Phase.CLEANUP })
public static short parseShort(String str)
  throws java.lang.NumberFormatException
```

Allocates no memory. Does not allow ”str” argument to escape local variables.

Throws NumberFormatException

```
@SCJAllowed
@SCJMaySelfSuspend(false)
@SCJMayAllocate({})
@SCJPhase({
    javax.safetycritical.annotate.Phase.STARTUP,
    javax.safetycritical.annotate.Phase.INITIALIZATION,
    javax.safetycritical.annotate.Phase.RUN,
    javax.safetycritical.annotate.Phase.CLEANUP })
public static short reverseBytes(short i)
```

Allocates no memory.

```
@SCJAllowed
@SCJMaySelfSuspend(false)
@SCJMayAllocate({})
@SCJPhase({
    javax.safetycritical.annotate.Phase.STARTUP,
    javax.safetycritical.annotate.Phase.INITIALIZATION,
    javax.safetycritical.annotate.Phase.RUN,
    javax.safetycritical.annotate.Phase.CLEANUP })
public short shortValue( )
```

Allocates no memory. Does not allow ”this” to escape local variables.

```
@SCJAllowed
@SCJMaySelfSuspend(false)
@SCJMayAllocate({javax.safetycritical.annotate.AllocationContext.CURRENT})
@SCJPhase({
    javax.safetycritical.annotate.Phase.STARTUP,
    javax.safetycritical.annotate.Phase.INITIALIZATION,
    javax.safetycritical.annotate.Phase.RUN,
    javax.safetycritical.annotate.Phase.CLEANUP })
public static java.lang.String toString(short v)
```

Allocates a String and associated internal ”structure” (e.g. char[]) in caller’s scope.

```
@SCJAllowed
@SCJMaySelfSuspend(false)
@SCJMayAllocate({javax.safetycritical.annotate.AllocationContext.CURRENT})
@SCJPhase({
    javax.safetycritical.annotate.Phase.STARTUP,
    javax.safetycritical.annotate.Phase.INITIALIZATION,
    javax.safetycritical.annotate.Phase.RUN,
    javax.safetycritical.annotate.Phase.CLEANUP })
public java.lang.String toString( )
```

Does not allow ”this” to escape local variables. Allocates a String and associated internal ”structure” (e.g. char[]) in caller’s scope. (Note: this semantics is desired for consistency with overridden implementation of Object.toString()).

```
@SCJAllowed
@SCJMaySelfSuspend(false)
@SCJMayAllocate({javax.safetycritical.annotate.AllocationContext.CURRENT})
@SCJPhase({
   javax.safetycritical.annotate.Phase.STARTUP,
   javax.safetycritical.annotate.Phase.INITIALIZATION,
   javax.safetycritical.annotate.Phase.RUN,
   javax.safetycritical.annotate.Phase.CLEANUP })
public static java.lang.Short valueOf(String str, int base)
  throws java.lang.NumberFormatException
```

Does not allow ”str” argument to escape local variables. Allocates an Integer in caller’s scope.

```
@SCJAllowed
@SCJMaySelfSuspend(false)
@SCJMayAllocate({javax.safetycritical.annotate.AllocationContext.CURRENT})
@SCJPhase({
   javax.safetycritical.annotate.Phase.STARTUP,
   javax.safetycritical.annotate.Phase.INITIALIZATION,
   javax.safetycritical.annotate.Phase.RUN,
   javax.safetycritical.annotate.Phase.CLEANUP })
public static java.lang.Short valueOf(String str)
  throws java.lang.NumberFormatException
```

Does not allow ”str” argument to escape local variables. Allocates an Integer in caller’s scope.

Throws NumberFormatException

```
@SCJAllowed
@SCJMaySelfSuspend(false)
@SCJMayAllocate({javax.safetycritical.annotate.AllocationContext.CURRENT})
@SCJPhase({
   javax.safetycritical.annotate.Phase.STARTUP,
   javax.safetycritical.annotate.Phase.INITIALIZATION,
   javax.safetycritical.annotate.Phase.RUN,
   javax.safetycritical.annotate.Phase.CLEANUP })
public static java.lang.Short valueOf(short val)
```

Allocates a Short in caller’s scope.

B.3.37 CLASS StackOverflowError

```
@SCJAllowed
public class StackOverflowError implements java.io.Serializable extends
    java.lang.VirtualMachineError
```

Thrown when a stack overflow occurs because an application recurses too deeply.

Constructors

```
@SCJAllowed
@SCJMaySelfSuspend(false)
@SCJMayAllocate({})
@SCJPhase({
   javax.safetycritical.annotate.Phase.STARTUP,
   javax.safetycritical.annotate.Phase.INITIALIZATION,
   javax.safetycritical.annotate.Phase.RUN,
   javax.safetycritical.annotate.Phase.CLEANUP })
public StackOverflowError( )
```

Shall not copy ”this” to any instance or static field.

Allocates an application- and implementation-dependent amount of memory in the current scope (to represent stack backtrace).

```
@SCJAllowed
@SCJMaySelfSuspend(false)
@SCJMayAllocate({})
@SCJPhase({
   javax.safetycritical.annotate.Phase.STARTUP,
   javax.safetycritical.annotate.Phase.INITIALIZATION,
   javax.safetycritical.annotate.Phase.RUN,
   javax.safetycritical.annotate.Phase.CLEANUP })
public StackOverflowError(String msg)
```

Shall not copy ”this” to any instance or static field. The scope containing the msg argument must enclose the scope containing ”this”. Otherwise, an IllegalAssignmentError will be thrown.

Allocates an application- and implementation-dependent amount of memory in the current scope (to represent stack backtrace).

B.3.38 CLASS StackTraceElement

```
@SCJAllowed
public class StackTraceElement extends java.lang.Object
```

An element in a stack trace, as returned by Throwable.getStackTrace(). Each element represents a single stack frame.

Constructors

```
@SCJAllowed
@SCJMaySelfSuspend(false)
@SCJMayAllocate({
    javax.safetycritical.annotate.AllocationContext.CURRENT,
    javax.safetycritical.annotate.AllocationContext.INNER,
    javax.safetycritical.annotate.AllocationContext.OUTER})
@SCJPhase({
    javax.safetycritical.annotate.Phase.STARTUP,
    javax.safetycritical.annotate.Phase.INITIALIZATION,
    javax.safetycritical.annotate.Phase.RUN,
    javax.safetycritical.annotate.Phase.CLEANUP })
public StackTraceElement(String declaringClass,
  String methodName,
  String fileName,
  int lineNumber)
```

Shall not copy ”this” to any instance or static field.

Methods

```
@SCJAllowed
@SCJMaySelfSuspend(false)
@SCJMayAllocate({
    javax.safetycritical.annotate.AllocationContext.CURRENT,
    javax.safetycritical.annotate.AllocationContext.INNER,
    javax.safetycritical.annotate.AllocationContext.OUTER})
@SCJPhase({
    javax.safetycritical.annotate.Phase.STARTUP,
    javax.safetycritical.annotate.Phase.INITIALIZATION,
    javax.safetycritical.annotate.Phase.RUN,
    javax.safetycritical.annotate.Phase.CLEANUP })
public boolean equals(Object obj)
```

Allocates no memory. Does not allow ”this” or ”obj” argument to escape local variables.

```
@SCJAllowed
@SCJMaySelfSuspend(false)
@SCJMayAllocate({
   javax.safetycritical.annotate.AllocationContext.CURRENT,
   javax.safetycritical.annotate.AllocationContext.INNER,
   javax.safetycritical.annotate.AllocationContext.OUTER})
@SCJPhase({
   javax.safetycritical.annotate.Phase.STARTUP,
   javax.safetycritical.annotate.Phase.INITIALIZATION,
   javax.safetycritical.annotate.Phase.RUN,
   javax.safetycritical.annotate.Phase.CLEANUP })
public java.lang.String getClassName( )
```

Performs no memory allocation. Returns a reference to the same String message that was supplied as an argument to the constructor, or null if the class name was not specified at construction time.

```
@SCJAllowed
@SCJMaySelfSuspend(false)
@SCJMayAllocate({
   javax.safetycritical.annotate.AllocationContext.CURRENT,
   javax.safetycritical.annotate.AllocationContext.INNER,
   javax.safetycritical.annotate.AllocationContext.OUTER})
@SCJPhase({
   javax.safetycritical.annotate.Phase.STARTUP,
   javax.safetycritical.annotate.Phase.INITIALIZATION,
   javax.safetycritical.annotate.Phase.RUN,
   javax.safetycritical.annotate.Phase.CLEANUP })
public java.lang.String getFileName( )
```

Performs no memory allocation. Returns a reference to the same String message that was supplied as an argument to the constructor, or null if the file name was not specified at construction time.

```
@SCJAllowed
@SCJMaySelfSuspend(false)
@SCJMayAllocate({
   javax.safetycritical.annotate.AllocationContext.CURRENT,
   javax.safetycritical.annotate.AllocationContext.INNER,
   javax.safetycritical.annotate.AllocationContext.OUTER})
@SCJPhase({
   javax.safetycritical.annotate.Phase.STARTUP,
   javax.safetycritical.annotate.Phase.INITIALIZATION,
   javax.safetycritical.annotate.Phase.RUN,
   javax.safetycritical.annotate.Phase.CLEANUP })
public int getLineNumber( )
```

Performs no memory allocation.

```
@SCJAllowed
@SCJMaySelfSuspend(false)
@SCJMayAllocate({
   javax.safetycritical.annotate.AllocationContext.CURRENT,
   javax.safetycritical.annotate.AllocationContext.INNER,
   javax.safetycritical.annotate.AllocationContext.OUTER})
@SCJPhase({
   javax.safetycritical.annotate.Phase.STARTUP,
   javax.safetycritical.annotate.Phase.INITIALIZATION,
   javax.safetycritical.annotate.Phase.RUN,
   javax.safetycritical.annotate.Phase.CLEANUP })
public java.lang.String getMethodName( )
```

Performs no memory allocation. Returns a reference to the same String message that was supplied as an argument to the constructor, or null if the method name was not specified at construction time.

```
@SCJAllowed
@SCJMaySelfSuspend(false)
@SCJMayAllocate({
   javax.safetycritical.annotate.AllocationContext.CURRENT,
   javax.safetycritical.annotate.AllocationContext.INNER,
   javax.safetycritical.annotate.AllocationContext.OUTER})
@SCJPhase({
   javax.safetycritical.annotate.Phase.STARTUP,
   javax.safetycritical.annotate.Phase.INITIALIZATION,
   javax.safetycritical.annotate.Phase.RUN,
   javax.safetycritical.annotate.Phase.CLEANUP })
public int hashCode( )
```

Allocates no memory. Does not allow ”this” to escape local variables.

```
@SCJAllowed
@SCJMaySelfSuspend(false)
@SCJMayAllocate({
   javax.safetycritical.annotate.AllocationContext.CURRENT,
   javax.safetycritical.annotate.AllocationContext.INNER,
   javax.safetycritical.annotate.AllocationContext.OUTER})
@SCJPhase({
   javax.safetycritical.annotate.Phase.STARTUP,
   javax.safetycritical.annotate.Phase.INITIALIZATION,
   javax.safetycritical.annotate.Phase.RUN,
   javax.safetycritical.annotate.Phase.CLEANUP })
public boolean isNativeMethod( )
```

Allocates no memory. Does not allow ”this” to escape local variables.

```
@SCJAllowed
@SCJMaySelfSuspend(false)
@SCJMayAllocate({
   javax.safetycritical.annotate.AllocationContext.CURRENT,
   javax.safetycritical.annotate.AllocationContext.INNER,
   javax.safetycritical.annotate.AllocationContext.OUTER})
@SCJPhase({
   javax.safetycritical.annotate.Phase.STARTUP,
   javax.safetycritical.annotate.Phase.INITIALIZATION,
   javax.safetycritical.annotate.Phase.RUN,
   javax.safetycritical.annotate.Phase.CLEANUP })
public java.lang.String toString( )
```

Does not allow ”this” to escape local variables. Allocates a String and associated internal ”structure” (e.g. char[]) in caller’s scope. (Note: this semantics is desired for consistency with overridden implementation of Object.toString()).

B.3.39 CLASS **StrictMath**

```
@SCJAllowed
public final class StrictMath extends java.lang.Object
```

The class StrictMath contains methods for performing basic numeric operations such as the elementary exponential, logarithm, square root, and trigonometric functions.

Fields

```
@SCJAllowed
public static final double E
```

```
@SCJAllowed
public static final double PI
```

Methods

```
@SCJAllowed
@SCJMaySelfSuspend(false)
@SCJMayAllocate({})
@SCJPhase({
   javax.safetycritical.annotate.Phase.STARTUP,
   javax.safetycritical.annotate.Phase.INITIALIZATION,
```

```
    javax.safetycritical.annotate.Phase.RUN,
    javax.safetycritical.annotate.Phase.CLEANUP })
public static double IEEEremainder(double f1, double f2)
```

Allocates no memory.

```
@SCJAllowed
@SCJMaySelfSuspend(false)
@SCJMayAllocate({})
@SCJPhase({
    javax.safetycritical.annotate.Phase.STARTUP,
    javax.safetycritical.annotate.Phase.INITIALIZATION,
    javax.safetycritical.annotate.Phase.RUN,
    javax.safetycritical.annotate.Phase.CLEANUP })
public static long abs(long a)
```

Allocates no memory.

```
@SCJAllowed
@SCJMaySelfSuspend(false)
@SCJMayAllocate({})
@SCJPhase({
    javax.safetycritical.annotate.Phase.STARTUP,
    javax.safetycritical.annotate.Phase.INITIALIZATION,
    javax.safetycritical.annotate.Phase.RUN,
    javax.safetycritical.annotate.Phase.CLEANUP })
public static double abs(double a)
```

Allocates no memory.

```
@SCJAllowed
@SCJMaySelfSuspend(false)
@SCJMayAllocate({})
@SCJPhase({
    javax.safetycritical.annotate.Phase.STARTUP,
    javax.safetycritical.annotate.Phase.INITIALIZATION,
    javax.safetycritical.annotate.Phase.RUN,
    javax.safetycritical.annotate.Phase.CLEANUP })
public static float abs(float a)
```

Allocates no memory.

```
@SCJAllowed
@SCJMaySelfSuspend(false)
@SCJMayAllocate({})
@SCJPhase({
    javax.safetycritical.annotate.Phase.STARTUP,
    javax.safetycritical.annotate.Phase.INITIALIZATION,
    javax.safetycritical.annotate.Phase.RUN,
    javax.safetycritical.annotate.Phase.CLEANUP })
public static int abs(int a)
```

Allocates no memory.

```
@SCJAllowed
@SCJMaySelfSuspend(false)
@SCJMayAllocate({})
@SCJPhase({
    javax.safetycritical.annotate.Phase.STARTUP,
    javax.safetycritical.annotate.Phase.INITIALIZATION,
    javax.safetycritical.annotate.Phase.RUN,
    javax.safetycritical.annotate.Phase.CLEANUP })
public static double acos(double a)
```

Allocates no memory.

```
@SCJAllowed
@SCJMaySelfSuspend(false)
@SCJMayAllocate({})
@SCJPhase({
    javax.safetycritical.annotate.Phase.STARTUP,
    javax.safetycritical.annotate.Phase.INITIALIZATION,
    javax.safetycritical.annotate.Phase.RUN,
    javax.safetycritical.annotate.Phase.CLEANUP })
public static double asin(double a)
```

Allocates no memory.

```
@SCJAllowed
@SCJMaySelfSuspend(false)
@SCJMayAllocate({})
@SCJPhase({
    javax.safetycritical.annotate.Phase.STARTUP,
    javax.safetycritical.annotate.Phase.INITIALIZATION,
    javax.safetycritical.annotate.Phase.RUN,
    javax.safetycritical.annotate.Phase.CLEANUP })
public static double atan(double a)
```

Allocates no memory.

```
@SCJAllowed
@SCJMaySelfSuspend(false)
@SCJMayAllocate({})
@SCJPhase({
    javax.safetycritical.annotate.Phase.STARTUP,
    javax.safetycritical.annotate.Phase.INITIALIZATION,
    javax.safetycritical.annotate.Phase.RUN,
    javax.safetycritical.annotate.Phase.CLEANUP })
public static double atan2(double a, double b)
```

Allocates no memory.

```
@SCJAllowed
@SCJMaySelfSuspend(false)
@SCJMayAllocate({})
@SCJPhase({
    javax.safetycritical.annotate.Phase.STARTUP,
    javax.safetycritical.annotate.Phase.INITIALIZATION,
    javax.safetycritical.annotate.Phase.RUN,
    javax.safetycritical.annotate.Phase.CLEANUP })
public static double cbrt(double a)
```

Allocates no memory.

```
@SCJAllowed
@SCJMaySelfSuspend(false)
@SCJMayAllocate({})
@SCJPhase({
    javax.safetycritical.annotate.Phase.STARTUP,
    javax.safetycritical.annotate.Phase.INITIALIZATION,
    javax.safetycritical.annotate.Phase.RUN,
    javax.safetycritical.annotate.Phase.CLEANUP })
public static double ceil(double a)
```

Allocates no memory.

```
@SCJAllowed
@SCJMaySelfSuspend(false)
@SCJMayAllocate({})
@SCJPhase({
    javax.safetycritical.annotate.Phase.STARTUP,
    javax.safetycritical.annotate.Phase.INITIALIZATION,
    javax.safetycritical.annotate.Phase.RUN,
    javax.safetycritical.annotate.Phase.CLEANUP })
public static double copySign(float magnitude, float sign)
```

Allocates no memory.

```
@SCJAllowed
@SCJMaySelfSuspend(false)
@SCJMayAllocate({})
@SCJPhase({
    javax.safetycritical.annotate.Phase.STARTUP,
    javax.safetycritical.annotate.Phase.INITIALIZATION,
    javax.safetycritical.annotate.Phase.RUN,
    javax.safetycritical.annotate.Phase.CLEANUP })
public static double copySign(double magnitude, double sign)
```

Allocates no memory.

```
@SCJAllowed
@SCJMaySelfSuspend(false)
@SCJMayAllocate({})
@SCJPhase({
    javax.safetycritical.annotate.Phase.STARTUP,
    javax.safetycritical.annotate.Phase.INITIALIZATION,
    javax.safetycritical.annotate.Phase.RUN,
    javax.safetycritical.annotate.Phase.CLEANUP })
public static double cos(double a)
```

Allocates no memory.

```
@SCJAllowed
@SCJMaySelfSuspend(false)
@SCJMayAllocate({})
@SCJPhase({
    javax.safetycritical.annotate.Phase.STARTUP,
    javax.safetycritical.annotate.Phase.INITIALIZATION,
    javax.safetycritical.annotate.Phase.RUN,
    javax.safetycritical.annotate.Phase.CLEANUP })
public static double cosh(double a)
```

Allocates no memory.

```
@SCJAllowed
@SCJMaySelfSuspend(false)
@SCJMayAllocate({})
@SCJPhase({
    javax.safetycritical.annotate.Phase.STARTUP,
    javax.safetycritical.annotate.Phase.INITIALIZATION,
    javax.safetycritical.annotate.Phase.RUN,
    javax.safetycritical.annotate.Phase.CLEANUP })
public static double exp(double a)
```

Allocates no memory.

```
@SCJAllowed
@SCJMaySelfSuspend(false)
@SCJMayAllocate({})
@SCJPhase({
    javax.safetycritical.annotate.Phase.STARTUP,
    javax.safetycritical.annotate.Phase.INITIALIZATION,
    javax.safetycritical.annotate.Phase.RUN,
    javax.safetycritical.annotate.Phase.CLEANUP })
public static double expm1(double a)
```

Allocates no memory.

```
@SCJAllowed
@SCJMaySelfSuspend(false)
@SCJMayAllocate({})
@SCJPhase({
    javax.safetycritical.annotate.Phase.STARTUP,
    javax.safetycritical.annotate.Phase.INITIALIZATION,
    javax.safetycritical.annotate.Phase.RUN,
    javax.safetycritical.annotate.Phase.CLEANUP })
public static double floor(double a)
```

Allocates no memory.

```
@SCJAllowed
@SCJMaySelfSuspend(false)
@SCJMayAllocate({})
@SCJPhase({
    javax.safetycritical.annotate.Phase.STARTUP,
    javax.safetycritical.annotate.Phase.INITIALIZATION,
    javax.safetycritical.annotate.Phase.RUN,
    javax.safetycritical.annotate.Phase.CLEANUP })
public static int getExponent(float a)
```

Allocates no memory.

```
@SCJAllowed
@SCJMaySelfSuspend(false)
@SCJMayAllocate({})
@SCJPhase({
    javax.safetycritical.annotate.Phase.STARTUP,
    javax.safetycritical.annotate.Phase.INITIALIZATION,
    javax.safetycritical.annotate.Phase.RUN,
    javax.safetycritical.annotate.Phase.CLEANUP })
public static int getExponent(double a)
```

Allocates no memory.

```
@SCJAllowed
@SCJMaySelfSuspend(false)
@SCJMayAllocate({})
@SCJPhase({
    javax.safetycritical.annotate.Phase.STARTUP,
    javax.safetycritical.annotate.Phase.INITIALIZATION,
    javax.safetycritical.annotate.Phase.RUN,
    javax.safetycritical.annotate.Phase.CLEANUP })
public static double hypot(double x, double y)
```

Allocates no memory.

```
@SCJAllowed
@SCJMaySelfSuspend(false)
@SCJMayAllocate({})
@SCJPhase({
    javax.safetycritical.annotate.Phase.STARTUP,
    javax.safetycritical.annotate.Phase.INITIALIZATION,
    javax.safetycritical.annotate.Phase.RUN,
    javax.safetycritical.annotate.Phase.CLEANUP })
public static double log(double a)
```

Allocates no memory.

```
@SCJAllowed
@SCJMaySelfSuspend(false)
@SCJMayAllocate({})
@SCJPhase({
    javax.safetycritical.annotate.Phase.STARTUP,
    javax.safetycritical.annotate.Phase.INITIALIZATION,
    javax.safetycritical.annotate.Phase.RUN,
    javax.safetycritical.annotate.Phase.CLEANUP })
public static double log10(double a)
```

Allocates no memory.

```
@SCJAllowed
@SCJMaySelfSuspend(false)
@SCJMayAllocate({})
@SCJPhase({
    javax.safetycritical.annotate.Phase.STARTUP,
    javax.safetycritical.annotate.Phase.INITIALIZATION,
    javax.safetycritical.annotate.Phase.RUN,
    javax.safetycritical.annotate.Phase.CLEANUP })
public static double log1p(double a)
```

Allocates no memory.

```
@SCJAllowed
@SCJMaySelfSuspend(false)
@SCJMayAllocate({})
@SCJPhase({
    javax.safetycritical.annotate.Phase.STARTUP,
    javax.safetycritical.annotate.Phase.INITIALIZATION,
    javax.safetycritical.annotate.Phase.RUN,
    javax.safetycritical.annotate.Phase.CLEANUP })
public static double max(double a, double b)
```

Allocates no memory.

```
@SCJAllowed
@SCJMaySelfSuspend(false)
@SCJMayAllocate({})
@SCJPhase({
    javax.safetycritical.annotate.Phase.STARTUP,
    javax.safetycritical.annotate.Phase.INITIALIZATION,
    javax.safetycritical.annotate.Phase.RUN,
    javax.safetycritical.annotate.Phase.CLEANUP })
public static long max(long a, long b)
```

Allocates no memory.

```
@SCJAllowed
@SCJMaySelfSuspend(false)
@SCJMayAllocate({})
@SCJPhase({
    javax.safetycritical.annotate.Phase.STARTUP,
    javax.safetycritical.annotate.Phase.INITIALIZATION,
    javax.safetycritical.annotate.Phase.RUN,
    javax.safetycritical.annotate.Phase.CLEANUP })
public static float max(float a, float b)
```

Allocates no memory.

```
@SCJAllowed
@SCJMaySelfSuspend(false)
@SCJMayAllocate({})
@SCJPhase({
    javax.safetycritical.annotate.Phase.STARTUP,
    javax.safetycritical.annotate.Phase.INITIALIZATION,
    javax.safetycritical.annotate.Phase.RUN,
    javax.safetycritical.annotate.Phase.CLEANUP })
public static int max(int a, int b)
```

Allocates no memory.

```
@SCJAllowed
@SCJMaySelfSuspend(false)
@SCJMayAllocate({})
@SCJPhase({
    javax.safetycritical.annotate.Phase.STARTUP,
    javax.safetycritical.annotate.Phase.INITIALIZATION,
    javax.safetycritical.annotate.Phase.RUN,
    javax.safetycritical.annotate.Phase.CLEANUP })
public static long min(long a, long b)
```

Allocates no memory.

```
@SCJAllowed
@SCJMaySelfSuspend(false)
@SCJMayAllocate({})
@SCJPhase({
    javax.safetycritical.annotate.Phase.STARTUP,
    javax.safetycritical.annotate.Phase.INITIALIZATION,
    javax.safetycritical.annotate.Phase.RUN,
    javax.safetycritical.annotate.Phase.CLEANUP })
public static double min(double a, double b)
```

Allocates no memory.

```
@SCJAllowed
@SCJMaySelfSuspend(false)
@SCJMayAllocate({})
@SCJPhase({
    javax.safetycritical.annotate.Phase.STARTUP,
    javax.safetycritical.annotate.Phase.INITIALIZATION,
    javax.safetycritical.annotate.Phase.RUN,
    javax.safetycritical.annotate.Phase.CLEANUP })
public static float min(float a, float b)
```

Allocates no memory.

```
@SCJAllowed
@SCJMaySelfSuspend(false)
@SCJMayAllocate({})
@SCJPhase({
    javax.safetycritical.annotate.Phase.STARTUP,
    javax.safetycritical.annotate.Phase.INITIALIZATION,
    javax.safetycritical.annotate.Phase.RUN,
    javax.safetycritical.annotate.Phase.CLEANUP })
public static int min(int a, int b)
```

Allocates no memory.

```
@SCJAllowed
@SCJMaySelfSuspend(false)
@SCJMayAllocate({})
@SCJPhase({
    javax.safetycritical.annotate.Phase.STARTUP,
    javax.safetycritical.annotate.Phase.INITIALIZATION,
    javax.safetycritical.annotate.Phase.RUN,
    javax.safetycritical.annotate.Phase.CLEANUP })
public static float nextAfter(float start, float direction)
```

Allocates no memory.

```
@SCJAllowed
@SCJMaySelfSuspend(false)
@SCJMayAllocate({})
@SCJPhase({
    javax.safetycritical.annotate.Phase.STARTUP,
    javax.safetycritical.annotate.Phase.INITIALIZATION,
    javax.safetycritical.annotate.Phase.RUN,
    javax.safetycritical.annotate.Phase.CLEANUP })
public static double nextAfter(double start, double direction)
```

Allocates no memory.

```
@SCJAllowed
@SCJMaySelfSuspend(false)
@SCJMayAllocate({})
@SCJPhase({
    javax.safetycritical.annotate.Phase.STARTUP,
    javax.safetycritical.annotate.Phase.INITIALIZATION,
    javax.safetycritical.annotate.Phase.RUN,
    javax.safetycritical.annotate.Phase.CLEANUP })
public static float nextUp(float d)
```

Allocates no memory.

```
@SCJAllowed
@SCJMaySelfSuspend(false)
@SCJMayAllocate({})
@SCJPhase({
    javax.safetycritical.annotate.Phase.STARTUP,
    javax.safetycritical.annotate.Phase.INITIALIZATION,
    javax.safetycritical.annotate.Phase.RUN,
    javax.safetycritical.annotate.Phase.CLEANUP })
public static double nextUp(double d)
```

Allocates no memory.

```
@SCJAllowed
@SCJMaySelfSuspend(false)
@SCJMayAllocate({})
@SCJPhase({
    javax.safetycritical.annotate.Phase.STARTUP,
    javax.safetycritical.annotate.Phase.INITIALIZATION,
    javax.safetycritical.annotate.Phase.RUN,
    javax.safetycritical.annotate.Phase.CLEANUP })
public static double pow(double a, double b)
```

Allocates no memory.

```
@SCJAllowed
@SCJMaySelfSuspend(false)
@SCJMayAllocate({})
@SCJPhase({
    javax.safetycritical.annotate.Phase.STARTUP,
    javax.safetycritical.annotate.Phase.INITIALIZATION,
    javax.safetycritical.annotate.Phase.RUN,
    javax.safetycritical.annotate.Phase.CLEANUP })
public static double random( )
```

Allocates no memory.

```
@SCJAllowed
@SCJMaySelfSuspend(false)
@SCJMayAllocate({})
@SCJPhase({
    javax.safetycritical.annotate.Phase.STARTUP,
    javax.safetycritical.annotate.Phase.INITIALIZATION,
    javax.safetycritical.annotate.Phase.RUN,
    javax.safetycritical.annotate.Phase.CLEANUP })
public static double rint(double a)
```

Allocates no memory.

```
@SCJAllowed
@SCJMaySelfSuspend(false)
@SCJMayAllocate({})
@SCJPhase({
    javax.safetycritical.annotate.Phase.STARTUP,
    javax.safetycritical.annotate.Phase.INITIALIZATION,
    javax.safetycritical.annotate.Phase.RUN,
    javax.safetycritical.annotate.Phase.CLEANUP })
public static long round(double a)
```

Allocates no memory.

```
@SCJAllowed
@SCJMaySelfSuspend(false)
@SCJMayAllocate({})
@SCJPhase({
    javax.safetycritical.annotate.Phase.STARTUP,
    javax.safetycritical.annotate.Phase.INITIALIZATION,
    javax.safetycritical.annotate.Phase.RUN,
    javax.safetycritical.annotate.Phase.CLEANUP })
public static int round(float a)
```

Allocates no memory.

```
@SCJAllowed
@SCJMaySelfSuspend(false)
@SCJMayAllocate({})
@SCJPhase({
    javax.safetycritical.annotate.Phase.STARTUP,
    javax.safetycritical.annotate.Phase.INITIALIZATION,
    javax.safetycritical.annotate.Phase.RUN,
    javax.safetycritical.annotate.Phase.CLEANUP })
public static float scalb(float f, int scaleFactor)
```

Allocates no memory.

```
@SCJAllowed
@SCJMaySelfSuspend(false)
@SCJMayAllocate({})
@SCJPhase({
    javax.safetycritical.annotate.Phase.STARTUP,
    javax.safetycritical.annotate.Phase.INITIALIZATION,
    javax.safetycritical.annotate.Phase.RUN,
    javax.safetycritical.annotate.Phase.CLEANUP })
public static double scalb(double d, int scaleFactor)
```

Allocates no memory.

```
@SCJAllowed
@SCJMaySelfSuspend(false)
@SCJMayAllocate({})
@SCJPhase({
    javax.safetycritical.annotate.Phase.STARTUP,
    javax.safetycritical.annotate.Phase.INITIALIZATION,
    javax.safetycritical.annotate.Phase.RUN,
    javax.safetycritical.annotate.Phase.CLEANUP })
public static float signum(float f)
```

Allocates no memory.

```
@SCJAllowed
@SCJMaySelfSuspend(false)
@SCJMayAllocate({})
@SCJPhase({
    javax.safetycritical.annotate.Phase.STARTUP,
    javax.safetycritical.annotate.Phase.INITIALIZATION,
    javax.safetycritical.annotate.Phase.RUN,
    javax.safetycritical.annotate.Phase.CLEANUP })
public static double signum(double d)
```

Allocates no memory.

```
@SCJAllowed
@SCJMaySelfSuspend(false)
@SCJMayAllocate({})
@SCJPhase({
    javax.safetycritical.annotate.Phase.STARTUP,
    javax.safetycritical.annotate.Phase.INITIALIZATION,
    javax.safetycritical.annotate.Phase.RUN,
    javax.safetycritical.annotate.Phase.CLEANUP })
public static double sin(double a)
```

Allocates no memory.

```
@SCJAllowed
@SCJMaySelfSuspend(false)
@SCJMayAllocate({})
@SCJPhase({
    javax.safetycritical.annotate.Phase.STARTUP,
    javax.safetycritical.annotate.Phase.INITIALIZATION,
    javax.safetycritical.annotate.Phase.RUN,
    javax.safetycritical.annotate.Phase.CLEANUP })
public static double sinh(double a)
```

Allocates no memory.

```
@SCJAllowed
@SCJMaySelfSuspend(false)
@SCJMayAllocate({})
@SCJPhase({
    javax.safetycritical.annotate.Phase.STARTUP,
    javax.safetycritical.annotate.Phase.INITIALIZATION,
    javax.safetycritical.annotate.Phase.RUN,
    javax.safetycritical.annotate.Phase.CLEANUP })
public static double sqrt(double a)
```

Allocates no memory.

```
@SCJAllowed
@SCJMaySelfSuspend(false)
@SCJMayAllocate({})
@SCJPhase({
    javax.safetycritical.annotate.Phase.STARTUP,
    javax.safetycritical.annotate.Phase.INITIALIZATION,
    javax.safetycritical.annotate.Phase.RUN,
    javax.safetycritical.annotate.Phase.CLEANUP })
public static double tan(double a)
```

Allocates no memory.

```
@SCJAllowed
@SCJMaySelfSuspend(false)
@SCJMayAllocate({})
@SCJPhase({
    javax.safetycritical.annotate.Phase.STARTUP,
    javax.safetycritical.annotate.Phase.INITIALIZATION,
    javax.safetycritical.annotate.Phase.RUN,
    javax.safetycritical.annotate.Phase.CLEANUP })
public static double tanh(double a)
```

Allocates no memory.

```
@SCJAllowed
@SCJMaySelfSuspend(false)
@SCJMayAllocate({})
@SCJPhase({
    javax.safetycritical.annotate.Phase.STARTUP,
    javax.safetycritical.annotate.Phase.INITIALIZATION,
    javax.safetycritical.annotate.Phase.RUN,
    javax.safetycritical.annotate.Phase.CLEANUP })
public static double toDegrees(double val)
```

Allocates no memory.

```
@SCJAllowed
@SCJMaySelfSuspend(false)
@SCJMayAllocate({})
@SCJPhase({
    javax.safetycritical.annotate.Phase.STARTUP,
    javax.safetycritical.annotate.Phase.INITIALIZATION,
    javax.safetycritical.annotate.Phase.RUN,
    javax.safetycritical.annotate.Phase.CLEANUP })
public static double toRadians(double val)
```

Allocates no memory.

```
@SCJAllowed
@SCJMaySelfSuspend(false)
@SCJMayAllocate({})
@SCJPhase({
   javax.safetycritical.annotate.Phase.STARTUP,
   javax.safetycritical.annotate.Phase.INITIALIZATION,
   javax.safetycritical.annotate.Phase.RUN,
   javax.safetycritical.annotate.Phase.CLEANUP })
public static float ulp(float d)
```

Allocates no memory.

```
@SCJAllowed
@SCJMaySelfSuspend(false)
@SCJMayAllocate({})
@SCJPhase({
   javax.safetycritical.annotate.Phase.STARTUP,
   javax.safetycritical.annotate.Phase.INITIALIZATION,
   javax.safetycritical.annotate.Phase.RUN,
   javax.safetycritical.annotate.Phase.CLEANUP })
public static double ulp(double d)
```

Allocates no memory.

B.3.40 CLASS String

```
@SCJAllowed
public final class String implements java.lang.CharSequence,
    java.lang.Comparable<T>, java.io.Serializable extends java.lang.Object
```

The String class represents character strings. All string literals in Java programs, such as ”abc”, are implemented as instances of this class.

Constructors

```
@SCJAllowed
@SCJMaySelfSuspend(false)
@SCJMayAllocate({javax.safetycritical.annotate.AllocationContext.CURRENT})
@SCJPhase({
   javax.safetycritical.annotate.Phase.STARTUP,
   javax.safetycritical.annotate.Phase.INITIALIZATION,
   javax.safetycritical.annotate.Phase.RUN,
   javax.safetycritical.annotate.Phase.CLEANUP })
public String( )
```

Does not allow ”this” argument to escape local variables.

```
@SCJAllowed
@SCJMaySelfSuspend(false)
@SCJMayAllocate({javax.safetycritical.annotate.AllocationContext.CURRENT})
@SCJPhase({
   javax.safetycritical.annotate.Phase.STARTUP,
   javax.safetycritical.annotate.Phase.INITIALIZATION,
   javax.safetycritical.annotate.Phase.RUN,
   javax.safetycritical.annotate.Phase.CLEANUP })
public String(byte [] b)
```

Does not allow ”this” or ”b” argument to escape local variables. Allocates internal structure to hold the contents of b within the same scope as ”this”.

```
@SCJAllowed
@SCJMaySelfSuspend(false)
@SCJMayAllocate({javax.safetycritical.annotate.AllocationContext.CURRENT})
@SCJPhase({
   javax.safetycritical.annotate.Phase.STARTUP,
   javax.safetycritical.annotate.Phase.INITIALIZATION,
   javax.safetycritical.annotate.Phase.RUN,
   javax.safetycritical.annotate.Phase.CLEANUP })
public String(String s)
```

Does not allow ”this” or ”s” argument to escape local variables. Allocates internal structure to hold the contents of s within the same scope as ”this”.

```
@SCJAllowed
@SCJMaySelfSuspend(false)
@SCJMayAllocate({javax.safetycritical.annotate.AllocationContext.CURRENT})
@SCJPhase({
   javax.safetycritical.annotate.Phase.STARTUP,
   javax.safetycritical.annotate.Phase.INITIALIZATION,
   javax.safetycritical.annotate.Phase.RUN,
   javax.safetycritical.annotate.Phase.CLEANUP })
public String(byte [] b, int offset, int length)
```

Does not allow ”this” or ”b” argument to escape local variables. Allocates internal structure to hold the contents of b within the same scope as ”this”.

```
@SCJAllowed
@SCJMaySelfSuspend(false)
@SCJMayAllocate({javax.safetycritical.annotate.AllocationContext.CURRENT})
```

```
@SCJPhase({
   javax.safetycritical.annotate.Phase.STARTUP,
   javax.safetycritical.annotate.Phase.INITIALIZATION,
   javax.safetycritical.annotate.Phase.RUN,
   javax.safetycritical.annotate.Phase.CLEANUP })
public String(char [] c)
```

Does not allow ”this” or ”c” argument to escape local variables. Allocates internal structure to hold the contents of c within the same scope as ”this”.

```
@SCJAllowed
@SCJMaySelfSuspend(false)
@SCJMayAllocate({})
@SCJPhase({
   javax.safetycritical.annotate.Phase.STARTUP,
   javax.safetycritical.annotate.Phase.INITIALIZATION,
   javax.safetycritical.annotate.Phase.RUN,
   javax.safetycritical.annotate.Phase.CLEANUP })
public String(StringBuilder b)
```

Allocates no memory.

Does not allow ”this” to escape local variables. Requires that argument ”b” reside in a scope that encloses the scope of ”this”. Builds a link from ”this” to the internal structure of argument b.

Note that the subset implementation of StringBuilder does not mutate existing buffer contents.

```
@SCJAllowed
@SCJMaySelfSuspend(false)
@SCJMayAllocate({javax.safetycritical.annotate.AllocationContext.CURRENT})
@SCJPhase({
   javax.safetycritical.annotate.Phase.STARTUP,
   javax.safetycritical.annotate.Phase.INITIALIZATION,
   javax.safetycritical.annotate.Phase.RUN,
   javax.safetycritical.annotate.Phase.CLEANUP })
public String(char [] c, int offset, int length)
```

Does not allow ”this” or ”c” argument to escape local variables. Allocates internal structure to hold the contents of c within the same scope as ”this”.

Methods

```
@SCJAllowed
@SCJMaySelfSuspend(false)
@SCJMayAllocate({})
@SCJPhase({
   javax.safetycritical.annotate.Phase.STARTUP,
   javax.safetycritical.annotate.Phase.INITIALIZATION,
   javax.safetycritical.annotate.Phase.RUN,
   javax.safetycritical.annotate.Phase.CLEANUP })
public char charAt(int index)
```

Allocates no memory. Does not allow ”this” to escape local variables.

```
@SCJAllowed
@SCJMaySelfSuspend(false)
@SCJMayAllocate({})
@SCJPhase({
   javax.safetycritical.annotate.Phase.STARTUP,
   javax.safetycritical.annotate.Phase.INITIALIZATION,
   javax.safetycritical.annotate.Phase.RUN,
   javax.safetycritical.annotate.Phase.CLEANUP })
public int compareTo(String str)
```

Allocates no memory. Does not allow ”this” or ”str” argument to escape local variables.

```
@SCJAllowed
@SCJMaySelfSuspend(false)
@SCJMayAllocate({})
@SCJPhase({
   javax.safetycritical.annotate.Phase.STARTUP,
   javax.safetycritical.annotate.Phase.INITIALIZATION,
   javax.safetycritical.annotate.Phase.RUN,
   javax.safetycritical.annotate.Phase.CLEANUP })
public int compareToIgnoreCase(String str)
```

Allocates no memory. Does not allow ”this” or ”str” argument to escape local variables.

```
@SCJAllowed
@SCJMaySelfSuspend(false)
@SCJMayAllocate({javax.safetycritical.annotate.AllocationContext.CURRENT})
@SCJPhase({
   javax.safetycritical.annotate.Phase.STARTUP,
   javax.safetycritical.annotate.Phase.INITIALIZATION,
   javax.safetycritical.annotate.Phase.RUN,
   javax.safetycritical.annotate.Phase.CLEANUP })
public java.lang.String concat(String arg)
```

Does not allow ”this” or ”str” argument to escape local variables. Allocates a String and internal structure to hold the catenation result in the caller’s scope.

```
@SCJAllowed
@SCJMaySelfSuspend(false)
@SCJMayAllocate({})
@SCJPhase({
   javax.safetycritical.annotate.Phase.STARTUP,
   javax.safetycritical.annotate.Phase.INITIALIZATION,
   javax.safetycritical.annotate.Phase.RUN,
   javax.safetycritical.annotate.Phase.CLEANUP })
public boolean contains(CharSequence arg)
```

Does not allow ”this” or ”str” argument to escape local variables.

```
@SCJAllowed
@SCJMaySelfSuspend(false)
@SCJMayAllocate({})
@SCJPhase({
   javax.safetycritical.annotate.Phase.STARTUP,
   javax.safetycritical.annotate.Phase.INITIALIZATION,
   javax.safetycritical.annotate.Phase.RUN,
   javax.safetycritical.annotate.Phase.CLEANUP })
public boolean contentEquals(CharSequence cs)
```

Does not allow ”this” or ”str” argument to escape local variables.

```
@SCJAllowed
@SCJMaySelfSuspend(false)
@SCJMayAllocate({})
@SCJPhase({
   javax.safetycritical.annotate.Phase.STARTUP,
   javax.safetycritical.annotate.Phase.INITIALIZATION,
   javax.safetycritical.annotate.Phase.RUN,
   javax.safetycritical.annotate.Phase.CLEANUP })
public final boolean endsWith(String suffix)
```

Allocates no memory. Does not allow ”this” or ”suffix” argument to escape local variables.

```
@SCJAllowed
@SCJMaySelfSuspend(false)
@SCJMayAllocate({})
@SCJPhase({
```

```
    javax.safetycritical.annotate.Phase.STARTUP,
    javax.safetycritical.annotate.Phase.INITIALIZATION,
    javax.safetycritical.annotate.Phase.RUN,
    javax.safetycritical.annotate.Phase.CLEANUP })
public boolean equals(Object obj)
```

Allocates no memory. Does not allow ”this” or ”obj” argument to escape local variables.

```
@SCJAllowed
@SCJMaySelfSuspend(false)
@SCJMayAllocate({})
@SCJPhase({
    javax.safetycritical.annotate.Phase.STARTUP,
    javax.safetycritical.annotate.Phase.INITIALIZATION,
    javax.safetycritical.annotate.Phase.RUN,
    javax.safetycritical.annotate.Phase.CLEANUP })
public boolean equalsIgnoreCase(String str)
```

Allocates no memory. Does not allow ”this” or ”str” argument to escape local variables.

```
@SCJAllowed
@SCJMaySelfSuspend(false)
@SCJMayAllocate({javax.safetycritical.annotate.AllocationContext.CURRENT})
@SCJPhase({
    javax.safetycritical.annotate.Phase.STARTUP,
    javax.safetycritical.annotate.Phase.INITIALIZATION,
    javax.safetycritical.annotate.Phase.RUN,
    javax.safetycritical.annotate.Phase.CLEANUP })
public byte[] getBytes( )
```

Does not allow ”this” to escape local variables. Allocates a byte array in the caller’s context.

```
@SCJAllowed
@SCJMaySelfSuspend(false)
@SCJMayAllocate({})
@SCJPhase({
    javax.safetycritical.annotate.Phase.STARTUP,
    javax.safetycritical.annotate.Phase.INITIALIZATION,
    javax.safetycritical.annotate.Phase.RUN,
    javax.safetycritical.annotate.Phase.CLEANUP })
public void getChars(int src_begin, int src_end, char [] dst, int dst_begin)
```

Allocates no memory. Does not allow ”this” or ”dst” argument to escape local variables.

```
@SCJAllowed
@SCJMaySelfSuspend(false)
@SCJMayAllocate({})
@SCJPhase({
   javax.safetycritical.annotate.Phase.STARTUP,
   javax.safetycritical.annotate.Phase.INITIALIZATION,
   javax.safetycritical.annotate.Phase.RUN,
   javax.safetycritical.annotate.Phase.CLEANUP })
public int hashCode( )
```

Allocates no memory. Does not allow ”this” to escape local variables.

```
@SCJAllowed
@SCJMaySelfSuspend(false)
@SCJMayAllocate({})
@SCJPhase({
   javax.safetycritical.annotate.Phase.STARTUP,
   javax.safetycritical.annotate.Phase.INITIALIZATION,
   javax.safetycritical.annotate.Phase.RUN,
   javax.safetycritical.annotate.Phase.CLEANUP })
public int indexOf(int ch, int from_index)
```

Allocates no memory. Does not allow ”this” to escape local variables.

```
@SCJAllowed
@SCJMaySelfSuspend(false)
@SCJMayAllocate({})
@SCJPhase({
   javax.safetycritical.annotate.Phase.STARTUP,
   javax.safetycritical.annotate.Phase.INITIALIZATION,
   javax.safetycritical.annotate.Phase.RUN,
   javax.safetycritical.annotate.Phase.CLEANUP })
public int indexOf(String str, int from_index)
```

Allocates no memory. Does not allow ”this” or ”str” argument to escape local variables.

```
@SCJAllowed
@SCJMaySelfSuspend(false)
@SCJMayAllocate({})
@SCJPhase({
```

```
    javax.safetycritical.annotate.Phase.STARTUP,
    javax.safetycritical.annotate.Phase.INITIALIZATION,
    javax.safetycritical.annotate.Phase.RUN,
    javax.safetycritical.annotate.Phase.CLEANUP })
public int indexOf(String str)
```

Allocates no memory. Does not allow ”this” or ”str” argument to escape local variables.

```
@SCJAllowed
@SCJMaySelfSuspend(false)
@SCJMayAllocate({})
@SCJPhase({
    javax.safetycritical.annotate.Phase.STARTUP,
    javax.safetycritical.annotate.Phase.INITIALIZATION,
    javax.safetycritical.annotate.Phase.RUN,
    javax.safetycritical.annotate.Phase.CLEANUP })
public int indexOf(int ch)
```

Allocates no memory. Does not allow ”this” to escape local variables.

```
@SCJAllowed
@SCJMaySelfSuspend(false)
@SCJMayAllocate({})
@SCJPhase({
    javax.safetycritical.annotate.Phase.STARTUP,
    javax.safetycritical.annotate.Phase.INITIALIZATION,
    javax.safetycritical.annotate.Phase.RUN,
    javax.safetycritical.annotate.Phase.CLEANUP })
public boolean isEmpty( )
```

Allocates no memory. Does not allow ”this” argument to escape local variables.

```
@SCJAllowed
@SCJMaySelfSuspend(false)
@SCJMayAllocate({})
@SCJPhase({
    javax.safetycritical.annotate.Phase.STARTUP,
    javax.safetycritical.annotate.Phase.INITIALIZATION,
    javax.safetycritical.annotate.Phase.RUN,
    javax.safetycritical.annotate.Phase.CLEANUP })
public int lastIndexOf(String str)
```

Allocates no memory. Does not allow ”this” or ”str” argument to escape local variables.

```
@SCJAllowed
@SCJMaySelfSuspend(false)
@SCJMayAllocate({})
@SCJPhase({
   javax.safetycritical.annotate.Phase.STARTUP,
   javax.safetycritical.annotate.Phase.INITIALIZATION,
   javax.safetycritical.annotate.Phase.RUN,
   javax.safetycritical.annotate.Phase.CLEANUP })
public int lastIndexOf(String str, int from_index)
```

Allocates no memory. Does not allow ”this” or ”str” argument to escape local variables.

```
@SCJAllowed
@SCJMaySelfSuspend(false)
@SCJMayAllocate({})
@SCJPhase({
   javax.safetycritical.annotate.Phase.STARTUP,
   javax.safetycritical.annotate.Phase.INITIALIZATION,
   javax.safetycritical.annotate.Phase.RUN,
   javax.safetycritical.annotate.Phase.CLEANUP })
public int lastIndexOf(int ch, int from_index)
```

Allocates no memory. Does not allow ”this” to escape local variables.

```
@SCJAllowed
@SCJMaySelfSuspend(false)
@SCJMayAllocate({})
@SCJPhase({
   javax.safetycritical.annotate.Phase.STARTUP,
   javax.safetycritical.annotate.Phase.INITIALIZATION,
   javax.safetycritical.annotate.Phase.RUN,
   javax.safetycritical.annotate.Phase.CLEANUP })
public int lastIndexOf(int ch)
```

Allocates no memory. Does not allow ”this” to escape local variables.

```
@SCJAllowed
@SCJMaySelfSuspend(false)
@SCJMayAllocate({})
@SCJPhase({
   javax.safetycritical.annotate.Phase.STARTUP,
   javax.safetycritical.annotate.Phase.INITIALIZATION,
   javax.safetycritical.annotate.Phase.RUN,
   javax.safetycritical.annotate.Phase.CLEANUP })
public int length( )
```

Allocates no memory. Does not allow ”this” to escape local variables.

```
@SCJAllowed
@SCJMaySelfSuspend(false)
@SCJMayAllocate({})
@SCJPhase({
   javax.safetycritical.annotate.Phase.STARTUP,
   javax.safetycritical.annotate.Phase.INITIALIZATION,
   javax.safetycritical.annotate.Phase.RUN,
   javax.safetycritical.annotate.Phase.CLEANUP })
public boolean regionMatches(int myoffset, String str, int offset, int len)
```

Allocates no memory. Does not allow ”this” or ”str” argument to escape local variables.

```
@SCJAllowed
@SCJMaySelfSuspend(false)
@SCJMayAllocate({})
@SCJPhase({
   javax.safetycritical.annotate.Phase.STARTUP,
   javax.safetycritical.annotate.Phase.INITIALIZATION,
   javax.safetycritical.annotate.Phase.RUN,
   javax.safetycritical.annotate.Phase.CLEANUP })
public boolean regionMatches(boolean ignore_case,
  int myoffset,
  String str,
  int offset,
  int len)
```

Allocates no memory. Does not allow ”this” or ”str” argument to escape local variables.

```
@SCJAllowed
@SCJMaySelfSuspend(false)
@SCJMayAllocate({javax.safetycritical.annotate.AllocationContext.CURRENT})
@SCJPhase({
   javax.safetycritical.annotate.Phase.STARTUP,
   javax.safetycritical.annotate.Phase.INITIALIZATION,
   javax.safetycritical.annotate.Phase.RUN,
   javax.safetycritical.annotate.Phase.CLEANUP })
public java.lang.String replace(CharSequence target, CharSequence replacement)
```

Does not allow ”this”, ”target”, or ”replacement” arguments to escape local variables. Allocates a String and internal structure to hold the result in the caller’s scope.

```
@SCJAllowed
@SCJMaySelfSuspend(false)
@SCJMayAllocate({javax.safetycritical.annotate.AllocationContext.CURRENT})
@SCJPhase({
   javax.safetycritical.annotate.Phase.STARTUP,
   javax.safetycritical.annotate.Phase.INITIALIZATION,
   javax.safetycritical.annotate.Phase.RUN,
   javax.safetycritical.annotate.Phase.CLEANUP })
public java.lang.String replace(char oldChar, char newChar)
```

Does not allow ”this” argument to escape local variables. Allocates a String and internal structure to hold the result in the caller’s scope.

```
@SCJAllowed
@SCJMaySelfSuspend(false)
@SCJMayAllocate({})
@SCJPhase({
   javax.safetycritical.annotate.Phase.STARTUP,
   javax.safetycritical.annotate.Phase.INITIALIZATION,
   javax.safetycritical.annotate.Phase.RUN,
   javax.safetycritical.annotate.Phase.CLEANUP })
public final boolean startsWith(String prefix, int toffset)
```

Allocates no memory. Does not allow ”this” or ”prefix” argument to escape local variables.

```
@SCJAllowed
@SCJMaySelfSuspend(false)
@SCJMayAllocate({})
@SCJPhase({
   javax.safetycritical.annotate.Phase.STARTUP,
   javax.safetycritical.annotate.Phase.INITIALIZATION,
   javax.safetycritical.annotate.Phase.RUN,
   javax.safetycritical.annotate.Phase.CLEANUP })
public final boolean startsWith(String prefix)
```

Allocates no memory. Does not allow ”this” or ”prefix” argument to escape local variables.

```
@SCJAllowed
@SCJMaySelfSuspend(false)
@SCJMayAllocate({javax.safetycritical.annotate.AllocationContext.CURRENT})
@SCJPhase({
   javax.safetycritical.annotate.Phase.STARTUP,
   javax.safetycritical.annotate.Phase.INITIALIZATION,
   javax.safetycritical.annotate.Phase.RUN,
   javax.safetycritical.annotate.Phase.CLEANUP })
public java.lang.String subSequence(int start, int end)
```

Allocates a String object in the caller's scope. Requires that "this" reside in a scope that encloses the caller's scope, since the the returned String retains a reference to the internal structure of "this" String.

```
@SCJAllowed
@SCJMaySelfSuspend(false)
@SCJMayAllocate({javax.safetycritical.annotate.AllocationContext.CURRENT})
@SCJPhase({
   javax.safetycritical.annotate.Phase.STARTUP,
   javax.safetycritical.annotate.Phase.INITIALIZATION,
   javax.safetycritical.annotate.Phase.RUN,
   javax.safetycritical.annotate.Phase.CLEANUP })
public java.lang.String substring(int begin_index, int end_index)
```

Allocates a String object in the caller's scope. Requires that "this" reside in a scope that encloses the caller's scope, since the the returned String retains a reference to the internal structure of "this" String.

```
@SCJAllowed
@SCJMaySelfSuspend(false)
@SCJMayAllocate({javax.safetycritical.annotate.AllocationContext.CURRENT})
@SCJPhase({
   javax.safetycritical.annotate.Phase.STARTUP,
   javax.safetycritical.annotate.Phase.INITIALIZATION,
   javax.safetycritical.annotate.Phase.RUN,
   javax.safetycritical.annotate.Phase.CLEANUP })
public java.lang.String substring(int begin_index)
```

Allocates a String object in the caller's scope. Requires that "this" reside in a scope that encloses the caller's scope, since the the returned String retains a reference to the internal structure of "this" String.

```
@SCJAllowed
@SCJMaySelfSuspend(false)
@SCJMayAllocate({javax.safetycritical.annotate.AllocationContext.CURRENT})
@SCJPhase({
   javax.safetycritical.annotate.Phase.STARTUP,
   javax.safetycritical.annotate.Phase.INITIALIZATION,
   javax.safetycritical.annotate.Phase.RUN,
   javax.safetycritical.annotate.Phase.CLEANUP })
public char[] toCharArray( )
```

Does not allow "this" to escape local variables. Allocates a char array to hold the result of this method in the caller's scope.

```
@SCJAllowed
@SCJMaySelfSuspend(false)
@SCJMayAllocate({javax.safetycritical.annotate.AllocationContext.CURRENT})
@SCJPhase({
   javax.safetycritical.annotate.Phase.STARTUP,
   javax.safetycritical.annotate.Phase.INITIALIZATION,
   javax.safetycritical.annotate.Phase.RUN,
   javax.safetycritical.annotate.Phase.CLEANUP })
public java.lang.String toLowerCase( )
```

Does not allow ”this” to escape local variables. Allocates a String and internal structure to hold the result of this method in the caller’s scope.

```
@SCJAllowed
@SCJMaySelfSuspend(false)
@SCJMayAllocate({javax.safetycritical.annotate.AllocationContext.CURRENT})
@SCJPhase({
   javax.safetycritical.annotate.Phase.STARTUP,
   javax.safetycritical.annotate.Phase.INITIALIZATION,
   javax.safetycritical.annotate.Phase.RUN,
   javax.safetycritical.annotate.Phase.CLEANUP })
public java.lang.String toString( )
```

Does not allow ”this” to escape local variables. Allocates a String and associated internal ”structure” (e.g. char[]) in caller’s scope. (Note: this semantics is desired for consistency with overridden implementation of Object.toString()).

```
@SCJAllowed
@SCJMaySelfSuspend(false)
@SCJMayAllocate({javax.safetycritical.annotate.AllocationContext.CURRENT})
@SCJPhase({
   javax.safetycritical.annotate.Phase.STARTUP,
   javax.safetycritical.annotate.Phase.INITIALIZATION,
   javax.safetycritical.annotate.Phase.RUN,
   javax.safetycritical.annotate.Phase.CLEANUP })
public java.lang.String toUpperCase( )
```

Does not allow ”this” to escape local variables. Allocates a String and internal structure to hold the result of this method in the caller’s scope.

```
@SCJAllowed
@SCJMaySelfSuspend(false)
@SCJMayAllocate({javax.safetycritical.annotate.AllocationContext.CURRENT})
@SCJPhase({
   javax.safetycritical.annotate.Phase.STARTUP,
```

```
    javax.safetycritical.annotate.Phase.INITIALIZATION,
    javax.safetycritical.annotate.Phase.RUN,
    javax.safetycritical.annotate.Phase.CLEANUP })
public final java.lang.String trim( )
```

Allocates a String object in the caller's scope. Requires that "this" reside in a scope that encloses the caller's scope, since the returned String retains a reference to the internal structure of "this" String.

```
@SCJAllowed
@SCJMaySelfSuspend(false)
@SCJMayAllocate({javax.safetycritical.annotate.AllocationContext.CURRENT})
@SCJPhase({
    javax.safetycritical.annotate.Phase.STARTUP,
    javax.safetycritical.annotate.Phase.INITIALIZATION,
    javax.safetycritical.annotate.Phase.RUN,
    javax.safetycritical.annotate.Phase.CLEANUP })
public static java.lang.String valueOf(float f)
```

Allocates a String and associated internal "structure" (e.g. char[]) in caller's scope.

```
@SCJAllowed
@SCJMaySelfSuspend(false)
@SCJMayAllocate({javax.safetycritical.annotate.AllocationContext.CURRENT})
@SCJPhase({
    javax.safetycritical.annotate.Phase.STARTUP,
    javax.safetycritical.annotate.Phase.INITIALIZATION,
    javax.safetycritical.annotate.Phase.RUN,
    javax.safetycritical.annotate.Phase.CLEANUP })
public static java.lang.String valueOf(int i)
```

Allocates a String and associated internal "structure" (e.g. char[]) in caller's scope.

```
@SCJAllowed
@SCJMaySelfSuspend(false)
@SCJMayAllocate({javax.safetycritical.annotate.AllocationContext.CURRENT})
@SCJPhase({
    javax.safetycritical.annotate.Phase.STARTUP,
    javax.safetycritical.annotate.Phase.INITIALIZATION,
    javax.safetycritical.annotate.Phase.RUN,
    javax.safetycritical.annotate.Phase.CLEANUP })
public static java.lang.String valueOf(long l)
```

Allocates a String and associated internal ”structure” (e.g. char[]) in caller’s scope.

```
@SCJAllowed
@SCJMaySelfSuspend(false)
@SCJMayAllocate({javax.safetycritical.annotate.AllocationContext.CURRENT})
@SCJPhase({
   javax.safetycritical.annotate.Phase.STARTUP,
   javax.safetycritical.annotate.Phase.INITIALIZATION,
   javax.safetycritical.annotate.Phase.RUN,
   javax.safetycritical.annotate.Phase.CLEANUP })
public static java.lang.String valueOf(Object o)
```

Allocates a String object in the caller’s scope.

```
@SCJAllowed
@SCJMaySelfSuspend(false)
@SCJMayAllocate({javax.safetycritical.annotate.AllocationContext.CURRENT})
@SCJPhase({
   javax.safetycritical.annotate.Phase.STARTUP,
   javax.safetycritical.annotate.Phase.INITIALIZATION,
   javax.safetycritical.annotate.Phase.RUN,
   javax.safetycritical.annotate.Phase.CLEANUP })
public static java.lang.String valueOf(char [] data)
```

Does not allow ”data” argument to escape local variables. Allocates a String and associated internal ”structure” (e.g. char[]) in caller’s scope.

```
@SCJAllowed
@SCJMaySelfSuspend(false)
@SCJMayAllocate({javax.safetycritical.annotate.AllocationContext.CURRENT})
@SCJPhase({
   javax.safetycritical.annotate.Phase.STARTUP,
   javax.safetycritical.annotate.Phase.INITIALIZATION,
   javax.safetycritical.annotate.Phase.RUN,
   javax.safetycritical.annotate.Phase.CLEANUP })
public static java.lang.String valueOf(char [] data, int offset, int count)
```

Does not allow ”data” argument to escape local variables. Allocates a String and associated internal ”structure” (e.g. char[]) in caller’s scope.

```
@SCJAllowed
@SCJMaySelfSuspend(false)
@SCJMayAllocate({javax.safetycritical.annotate.AllocationContext.CURRENT})
```

```
@SCJPhase({
    javax.safetycritical.annotate.Phase.STARTUP,
    javax.safetycritical.annotate.Phase.INITIALIZATION,
    javax.safetycritical.annotate.Phase.RUN,
    javax.safetycritical.annotate.Phase.CLEANUP })
public static java.lang.String valueOf(double d)
```

Allocates a String and associated internal ”structure” (e.g. char[]) in caller’s scope.

```
@SCJAllowed
@SCJMaySelfSuspend(false)
@SCJMayAllocate({javax.safetycritical.annotate.AllocationContext.CURRENT})
@SCJPhase({
    javax.safetycritical.annotate.Phase.STARTUP,
    javax.safetycritical.annotate.Phase.INITIALIZATION,
    javax.safetycritical.annotate.Phase.RUN,
    javax.safetycritical.annotate.Phase.CLEANUP })
public static java.lang.String valueOf(char c)
```

Allocates a String and associated internal ”structure” (e.g. char[]) in caller’s scope.

```
@SCJAllowed
@SCJMaySelfSuspend(false)
@SCJMayAllocate({})
@SCJPhase({
    javax.safetycritical.annotate.Phase.STARTUP,
    javax.safetycritical.annotate.Phase.INITIALIZATION,
    javax.safetycritical.annotate.Phase.RUN,
    javax.safetycritical.annotate.Phase.CLEANUP })
public static java.lang.String valueOf(boolean b)
```

Allocates no memory. Returns a preallocated String residing in the scope of the String class’s ClassLoader.

B.3.41 CLASS StringBuilder

```
@SCJAllowed
public final class StringBuilder implements java.lang.Appendable,
    java.lang.CharSequence, java.io.Serializable extends java.lang.Object
```

A mutable sequence of characters.

Constructors

```
@SCJAllowed
@SCJMaySelfSuspend(true)
@SCJMayAllocate({javax.safetycritical.annotate.AllocationContext.CURRENT})
@SCJPhase({
   javax.safetycritical.annotate.Phase.STARTUP,
   javax.safetycritical.annotate.Phase.INITIALIZATION,
   javax.safetycritical.annotate.Phase.RUN,
   javax.safetycritical.annotate.Phase.CLEANUP })
public StringBuilder( )
```

Does not allow ”this” to escape local variables. Allocates internal structure of sufficient size to represent 16 characters in the scope of ”this”.

```
@SCJAllowed
@SCJMaySelfSuspend(true)
@SCJMayAllocate({javax.safetycritical.annotate.AllocationContext.CURRENT})
@SCJPhase({
   javax.safetycritical.annotate.Phase.STARTUP,
   javax.safetycritical.annotate.Phase.INITIALIZATION,
   javax.safetycritical.annotate.Phase.RUN,
   javax.safetycritical.annotate.Phase.CLEANUP })
public StringBuilder(int length)
```

Does not allow ”this” to escape local variables. Allocates internal structure of sufficient size to represent length characters within the scope of ”this”.

```
@SCJAllowed
@SCJMaySelfSuspend(true)
@SCJMayAllocate({javax.safetycritical.annotate.AllocationContext.CURRENT})
@SCJPhase({
   javax.safetycritical.annotate.Phase.STARTUP,
   javax.safetycritical.annotate.Phase.INITIALIZATION,
   javax.safetycritical.annotate.Phase.RUN,
   javax.safetycritical.annotate.Phase.CLEANUP })
public StringBuilder(String str)
```

Does not allow ”this” to escape local variables. Allocates a character internal structure of sufficient size to represent str.length() + 16 characters within the scope of ”this”.

```
@SCJAllowed
@SCJMaySelfSuspend(true)
@SCJMayAllocate({javax.safetycritical.annotate.AllocationContext.CURRENT})
@SCJPhase({
   javax.safetycritical.annotate.Phase.STARTUP,
```

```
    javax.safetycritical.annotate.Phase.INITIALIZATION,
    javax.safetycritical.annotate.Phase.RUN,
    javax.safetycritical.annotate.Phase.CLEANUP })
public StringBuilder(CharSequence seq)
```

Does not allow ”this” to escape local variables. Allocates a character internal structure of sufficient size to represent seq.length() + 16 characters within the scope of ”this”.

Methods

```
@SCJAllowed
@SCJMaySelfSuspend(true)
@SCJMayAllocate({javax.safetycritical.annotate.AllocationContext.THIS})
@SCJPhase({
    javax.safetycritical.annotate.Phase.STARTUP,
    javax.safetycritical.annotate.Phase.INITIALIZATION,
    javax.safetycritical.annotate.Phase.RUN,
    javax.safetycritical.annotate.Phase.CLEANUP })
public java.lang.StringBuilder append(char c)
```

Does not allow ”this” to escape local variables. If expansion of ”this” StringBuilder’s internal character buffer is necessary, a new char array is allocated within the scope of ”this”. The new array will be twice the length of the existing array, plus 1.

```
@SCJAllowed
@SCJMaySelfSuspend(true)
@SCJMayAllocate({javax.safetycritical.annotate.AllocationContext.THIS})
@SCJPhase({
    javax.safetycritical.annotate.Phase.STARTUP,
    javax.safetycritical.annotate.Phase.INITIALIZATION,
    javax.safetycritical.annotate.Phase.RUN,
    javax.safetycritical.annotate.Phase.CLEANUP })
public java.lang.StringBuilder append(char [] buf, int offset, int length)
```

Does not allow ”this” or ”buf” to escape local variables. If expansion of ”this” StringBuilder’s internal character buffer is necessary, a new char array is allocated within the scope of ”this”. The new array will be twice the length of the existing array, plus 1.

```
@SCJAllowed
@SCJMaySelfSuspend(true)
@SCJMayAllocate({javax.safetycritical.annotate.AllocationContext.THIS})
```

```
@SCJPhase({
   javax.safetycritical.annotate.Phase.STARTUP,
   javax.safetycritical.annotate.Phase.INITIALIZATION,
   javax.safetycritical.annotate.Phase.RUN,
   javax.safetycritical.annotate.Phase.CLEANUP })
public java.lang.StringBuilder append(CharSequence cs, int start, int end)
```

Does not allow ”this” or argument ”cs” to escape local variables. If expansion of ”this” StringBuilder’s internal character buffer is necessary, a new char array is allocated within the scope of ”this”. The new array will be twice the length of the existing array, plus 1.

```
@SCJAllowed
@SCJMaySelfSuspend(true)
@SCJMayAllocate({javax.safetycritical.annotate.AllocationContext.THIS})
@SCJPhase({
   javax.safetycritical.annotate.Phase.STARTUP,
   javax.safetycritical.annotate.Phase.INITIALIZATION,
   javax.safetycritical.annotate.Phase.RUN,
   javax.safetycritical.annotate.Phase.CLEANUP })
public java.lang.StringBuilder append(float f)
```

Does not allow ”this” to escape local variables. If expansion of ”this” StringBuilder’s internal character buffer is necessary, a new char array is allocated within the scope of ”this”. The new array will be twice the length of the existing array, plus 1.

```
@SCJAllowed
@SCJMaySelfSuspend(true)
@SCJMayAllocate({javax.safetycritical.annotate.AllocationContext.THIS})
@SCJPhase({
   javax.safetycritical.annotate.Phase.STARTUP,
   javax.safetycritical.annotate.Phase.INITIALIZATION,
   javax.safetycritical.annotate.Phase.RUN,
   javax.safetycritical.annotate.Phase.CLEANUP })
public java.lang.StringBuilder append(long l)
```

Does not allow ”this” to escape local variables. If expansion of ”this” StringBuilder’s internal character buffer is necessary, a new char array is allocated within the scope of ”this”. The new array will be twice the length of the existing array, plus 1.

```
@SCJAllowed
@SCJMaySelfSuspend(true)
@SCJMayAllocate({javax.safetycritical.annotate.AllocationContext.THIS})
@SCJPhase({
   javax.safetycritical.annotate.Phase.STARTUP,
   javax.safetycritical.annotate.Phase.INITIALIZATION,
   javax.safetycritical.annotate.Phase.RUN,
   javax.safetycritical.annotate.Phase.CLEANUP })
public java.lang.StringBuilder append(String s)
```

Does not allow ”this” or argument ”s” to escape local variables. If expansion of ”this” StringBuilder’s internal character buffer is necessary, a new char array is allocated within the scope of ”this”. The new array will be twice the length of the existing array, plus 1.

```
@SCJAllowed
@SCJMaySelfSuspend(true)
@SCJMayAllocate({javax.safetycritical.annotate.AllocationContext.THIS})
@SCJPhase({
   javax.safetycritical.annotate.Phase.STARTUP,
   javax.safetycritical.annotate.Phase.INITIALIZATION,
   javax.safetycritical.annotate.Phase.RUN,
   javax.safetycritical.annotate.Phase.CLEANUP })
public java.lang.StringBuilder append(Object o)
```

Does not allow ”this” to escape local variables. If expansion of ”this” StringBuilder’s internal character buffer is necessary, a new char array is allocated within the scope of ”this”. The new array will be twice the length of the existing array, plus 1.

Requires that argument ”o” reside in a scope that encloses ”this”

```
@SCJAllowed
@SCJMaySelfSuspend(true)
@SCJMayAllocate({javax.safetycritical.annotate.AllocationContext.THIS})
@SCJPhase({
   javax.safetycritical.annotate.Phase.STARTUP,
   javax.safetycritical.annotate.Phase.INITIALIZATION,
   javax.safetycritical.annotate.Phase.RUN,
   javax.safetycritical.annotate.Phase.CLEANUP })
public java.lang.StringBuilder append(int i)
```

Does not allow ”this” to escape local variables. If expansion of ”this” StringBuilder’s internal character buffer is necessary, a new char array is allocated within the scope of ”this”. The new array will be twice the length of the existing array, plus 1.

```
@SCJAllowed
@SCJMaySelfSuspend(true)
@SCJMayAllocate({javax.safetycritical.annotate.AllocationContext.THIS})
@SCJPhase({
   javax.safetycritical.annotate.Phase.STARTUP,
   javax.safetycritical.annotate.Phase.INITIALIZATION,
   javax.safetycritical.annotate.Phase.RUN,
   javax.safetycritical.annotate.Phase.CLEANUP })
public java.lang.StringBuilder append(double d)
```

Does not allow ”this” to escape local variables. If expansion of ”this” StringBuilder’s internal character buffer is necessary, a new char array is allocated within the scope of ”this”. The new array will be twice the length of the existing array, plus 1.

```
@SCJAllowed
@SCJMaySelfSuspend(true)
@SCJMayAllocate({javax.safetycritical.annotate.AllocationContext.THIS})
@SCJPhase({
   javax.safetycritical.annotate.Phase.STARTUP,
   javax.safetycritical.annotate.Phase.INITIALIZATION,
   javax.safetycritical.annotate.Phase.RUN,
   javax.safetycritical.annotate.Phase.CLEANUP })
public java.lang.StringBuilder append(CharSequence cs)
```

Does not allow ”this” or argument ”cs” to escape local variables. If expansion of ”this” StringBuilder’s internal character buffer is necessary, a new char array is allocated within the scope of ”this”. The new array will be twice the length of the existing array, plus 1.

```
@SCJAllowed
@SCJMaySelfSuspend(true)
@SCJMayAllocate({javax.safetycritical.annotate.AllocationContext.THIS})
@SCJPhase({
   javax.safetycritical.annotate.Phase.STARTUP,
   javax.safetycritical.annotate.Phase.INITIALIZATION,
   javax.safetycritical.annotate.Phase.RUN,
   javax.safetycritical.annotate.Phase.CLEANUP })
public java.lang.StringBuilder append(char [] buf)
```

Does not allow ”this” or ”buf” to escape local variables. If expansion of ”this” StringBuilder’s internal character buffer is necessary, a new char array is allocated within the scope of ”this”. The new array will be twice the length of the existing array, plus 1.

```
@SCJAllowed
@SCJMaySelfSuspend(true)
@SCJMayAllocate({javax.safetycritical.annotate.AllocationContext.THIS})
@SCJPhase({
   javax.safetycritical.annotate.Phase.STARTUP,
   javax.safetycritical.annotate.Phase.INITIALIZATION,
   javax.safetycritical.annotate.Phase.RUN,
   javax.safetycritical.annotate.Phase.CLEANUP })
public java.lang.StringBuilder append(boolean b)
```

Does not allow ”this” to escape local variables. If expansion of ”this” StringBuilder’s internal character buffer is necessary, a new char array is allocated within the scope of ”this”. The new array will be twice the length of the existing array, plus 1.

```
@SCJAllowed
@SCJMaySelfSuspend(true)
@SCJMayAllocate({})
@SCJPhase({
   javax.safetycritical.annotate.Phase.STARTUP,
   javax.safetycritical.annotate.Phase.INITIALIZATION,
   javax.safetycritical.annotate.Phase.RUN,
   javax.safetycritical.annotate.Phase.CLEANUP })
public int capacity( )
```

Allocates no memory. Does not allow ”this” to escape local variables.

```
@SCJAllowed
@SCJMaySelfSuspend(true)
@SCJMayAllocate({})
@SCJPhase({
   javax.safetycritical.annotate.Phase.STARTUP,
   javax.safetycritical.annotate.Phase.INITIALIZATION,
   javax.safetycritical.annotate.Phase.RUN,
   javax.safetycritical.annotate.Phase.CLEANUP })
public char charAt(int index)
```

Allocates no memory. Does not allow ”this” to escape local variables.

```
@SCJAllowed
@SCJMaySelfSuspend(true)
@SCJMayAllocate({javax.safetycritical.annotate.AllocationContext.THIS})
@SCJPhase({
   javax.safetycritical.annotate.Phase.STARTUP,
   javax.safetycritical.annotate.Phase.INITIALIZATION,
   javax.safetycritical.annotate.Phase.RUN,
   javax.safetycritical.annotate.Phase.CLEANUP })
public void ensureCapacity(int minimum_capacity)
```

Does not allow ”this” to escape local variables. If expansion of ”this” StringBuilder’s internal character buffer is necessary, a new char array is allocated within the scope of ”this”. The new array will be twice the length of the existing array, plus 1.

```
@SCJAllowed
@SCJMaySelfSuspend(true)
@SCJMayAllocate({})
@SCJPhase({
   javax.safetycritical.annotate.Phase.STARTUP,
   javax.safetycritical.annotate.Phase.INITIALIZATION,
   javax.safetycritical.annotate.Phase.RUN,
   javax.safetycritical.annotate.Phase.CLEANUP })
public void getChars(int srcBegin, int srcEnd, char [] dst, int dstBegin)
```

Does not allow ”this” or ”dst” to escape local variables.

```
@SCJAllowed
@SCJMaySelfSuspend(true)
@SCJMayAllocate({})
@SCJPhase({
   javax.safetycritical.annotate.Phase.STARTUP,
   javax.safetycritical.annotate.Phase.INITIALIZATION,
   javax.safetycritical.annotate.Phase.RUN,
   javax.safetycritical.annotate.Phase.CLEANUP })
public void indexOf(String str, int fromIndex)
```

Does not allow ”this” or ”dst” to escape local variables.

```
@SCJAllowed
@SCJMaySelfSuspend(true)
@SCJMayAllocate({})
@SCJPhase({
   javax.safetycritical.annotate.Phase.STARTUP,
   javax.safetycritical.annotate.Phase.INITIALIZATION,
   javax.safetycritical.annotate.Phase.RUN,
   javax.safetycritical.annotate.Phase.CLEANUP })
public void indexOf(String str)
```

Does not allow ”this” or ”dst” to escape local variables.

```
@SCJAllowed
@SCJMaySelfSuspend(true)
@SCJMayAllocate({})
```

```
@SCJPhase({
    javax.safetycritical.annotate.Phase.STARTUP,
    javax.safetycritical.annotate.Phase.INITIALIZATION,
    javax.safetycritical.annotate.Phase.RUN,
    javax.safetycritical.annotate.Phase.CLEANUP })
public void lastIndexOf(String str, int fromIndex)
```

Does not allow ”this” or ”dst” to escape local variables.

```
@SCJAllowed
@SCJMaySelfSuspend(true)
@SCJMayAllocate({})
@SCJPhase({
    javax.safetycritical.annotate.Phase.STARTUP,
    javax.safetycritical.annotate.Phase.INITIALIZATION,
    javax.safetycritical.annotate.Phase.RUN,
    javax.safetycritical.annotate.Phase.CLEANUP })
public void lastIndexOf(String str)
```

Does not allow ”this” or ”dst” to escape local variables.

```
@SCJAllowed
@SCJMaySelfSuspend(true)
@SCJMayAllocate({})
@SCJPhase({
    javax.safetycritical.annotate.Phase.STARTUP,
    javax.safetycritical.annotate.Phase.INITIALIZATION,
    javax.safetycritical.annotate.Phase.RUN,
    javax.safetycritical.annotate.Phase.CLEANUP })
public int length( )
```

Allocates no memory. Does not allow ”this” to escape local variables.

```
@SCJAllowed
@SCJMaySelfSuspend(true)
@SCJMayAllocate({})
@SCJPhase({
    javax.safetycritical.annotate.Phase.STARTUP,
    javax.safetycritical.annotate.Phase.INITIALIZATION,
    javax.safetycritical.annotate.Phase.RUN,
    javax.safetycritical.annotate.Phase.CLEANUP })
public void setLength(int new_length)
  throws java.lang.IndexOutOfBoundsException
```

Does not allow ”this” to escape local variables.

Throws IndexOutOfBoundsException

```
@SCJAllowed
@SCJMaySelfSuspend(true)
@SCJMayAllocate({javax.safetycritical.annotate.AllocationContext.CURRENT})
@SCJPhase({
   javax.safetycritical.annotate.Phase.STARTUP,
   javax.safetycritical.annotate.Phase.INITIALIZATION,
   javax.safetycritical.annotate.Phase.RUN,
   javax.safetycritical.annotate.Phase.CLEANUP })
public java.lang.CharSequence subSequence(int start, int end)
```

Does not allow ”this” to escape local variables. Allocates a String in caller’s scope.

```
@SCJAllowed
@SCJMaySelfSuspend(true)
@SCJMayAllocate({javax.safetycritical.annotate.AllocationContext.CURRENT})
@SCJPhase({
   javax.safetycritical.annotate.Phase.STARTUP,
   javax.safetycritical.annotate.Phase.INITIALIZATION,
   javax.safetycritical.annotate.Phase.RUN,
   javax.safetycritical.annotate.Phase.CLEANUP })
public java.lang.String toString( )
```

Does not allow ”this” to escape local variables. Allocates a String in caller’s scope.

B.3.42 CLASS StringIndexOutOfBoundsException

```
@SCJAllowed
public class StringIndexOutOfBoundsException implements java.io.Serializable
    extends java.lang.IndexOutOfBoundsException
```

Thrown by String methods to indicate that an index is either negative or greater than the size of the string.

Constructors

```
@SCJAllowed
@SCJMaySelfSuspend(false)
@SCJMayAllocate({javax.safetycritical.annotate.AllocationContext.CURRENT})
@SCJPhase({
   javax.safetycritical.annotate.Phase.STARTUP,
```

```
    javax.safetycritical.annotate.Phase.INITIALIZATION,
    javax.safetycritical.annotate.Phase.RUN,
    javax.safetycritical.annotate.Phase.CLEANUP })
public StringIndexOutOfBoundsException( )
```

Shall not copy ”this” to any instance or static field.

Allocates an application- and implementation-dependent amount of memory in the current scope (to represent stack backtrace).

```
@SCJAllowed
@SCJMaySelfSuspend(false)
@SCJMayAllocate({javax.safetycritical.annotate.AllocationContext.CURRENT})
@SCJPhase({
    javax.safetycritical.annotate.Phase.STARTUP,
    javax.safetycritical.annotate.Phase.INITIALIZATION,
    javax.safetycritical.annotate.Phase.RUN,
    javax.safetycritical.annotate.Phase.CLEANUP })
public StringIndexOutOfBoundsException(int index)
```

Shall not copy ”this” to any instance or static field.

Allocates an application- and implementation-dependent amount of memory in the current scope (to represent stack backtrace).

```
@SCJAllowed
@SCJMaySelfSuspend(false)
@SCJMayAllocate({javax.safetycritical.annotate.AllocationContext.CURRENT})
@SCJPhase({
    javax.safetycritical.annotate.Phase.STARTUP,
    javax.safetycritical.annotate.Phase.INITIALIZATION,
    javax.safetycritical.annotate.Phase.RUN,
    javax.safetycritical.annotate.Phase.CLEANUP })
public StringIndexOutOfBoundsException(String msg)
```

Shall not copy ”this” to any instance or static field. The scope containing the msg argument must enclose the scope containing ”this”. Otherwise, an IllegalAssignmentError will be thrown.

Allocates an application- and implementation-dependent amount of memory in the current scope (to represent stack backtrace).

B.3.43 CLASS System

```
@SCJAllowed
public final class System extends java.lang.Object
```

The System class contains several useful class fields and methods. It cannot be instantiated.

Constructors

```
@SCJAllowed
@SCJMaySelfSuspend(false)
@SCJMayAllocate({javax.safetycritical.annotate.AllocationContext.CURRENT})
@SCJPhase({
    javax.safetycritical.annotate.Phase.STARTUP,
    javax.safetycritical.annotate.Phase.INITIALIZATION,
    javax.safetycritical.annotate.Phase.RUN,
    javax.safetycritical.annotate.Phase.CLEANUP })
protected System( )
```

Allocates no memory.

Methods

```
@SCJAllowed
@SCJMaySelfSuspend(false)
@SCJMayAllocate({})
@SCJPhase({
    javax.safetycritical.annotate.Phase.STARTUP,
    javax.safetycritical.annotate.Phase.INITIALIZATION,
    javax.safetycritical.annotate.Phase.RUN,
    javax.safetycritical.annotate.Phase.CLEANUP })
public static void arraycopy(Object src,
  int srcPos,
  Object dest,
  int destPos,
  int length)
```

Allocates no memory. Does not allow ”src” or ”dest” arguments to escape local variables. Allocates no memory.

Requires that the contents of array src enclose array dest.

```
@SCJAllowed
@SCJMaySelfSuspend(false)
@SCJMayAllocate({})
```

```
@SCJPhase({
    javax.safetycritical.annotate.Phase.STARTUP,
    javax.safetycritical.annotate.Phase.INITIALIZATION,
    javax.safetycritical.annotate.Phase.RUN,
    javax.safetycritical.annotate.Phase.CLEANUP })
public static long currentTimeMillis( )
```

Allocates no memory.

```
@SCJAllowed
@SCJMaySelfSuspend(false)
@SCJMayAllocate({})
@SCJPhase({
    javax.safetycritical.annotate.Phase.STARTUP,
    javax.safetycritical.annotate.Phase.INITIALIZATION,
    javax.safetycritical.annotate.Phase.RUN,
    javax.safetycritical.annotate.Phase.CLEANUP })
public static void exit(int code)
```

Allocates no memory.

```
@SCJAllowed
@SCJMaySelfSuspend(false)
@SCJMayAllocate({})
@SCJPhase({
    javax.safetycritical.annotate.Phase.STARTUP,
    javax.safetycritical.annotate.Phase.INITIALIZATION,
    javax.safetycritical.annotate.Phase.RUN,
    javax.safetycritical.annotate.Phase.CLEANUP })
public static java.lang.String getProperty(String key, String default_value)
```

Allocates no memory.

Unlike traditional J2SE, this method shall not cause a set of system properties to be created and initialized if not already existing. Any necessary initialization shall occur during system startup.

returns The value of the property associated with key, or the value of default_value if no property is associated with key. The value returned resides in immortal memory, or it is the value of default.

```
@SCJAllowed
@SCJMaySelfSuspend(false)
@SCJMayAllocate({})
@SCJPhase({
    javax.safetycritical.annotate.Phase.STARTUP,
```

```
    javax.safetycritical.annotate.Phase.INITIALIZATION,
    javax.safetycritical.annotate.Phase.RUN,
    javax.safetycritical.annotate.Phase.CLEANUP })
public static java.lang.String getProperty(String key)
```

Allocates no memory.

Unlike traditional J2SE, this method shall not cause a set of system properties to be created and initialized if not already existing. Any necessary initialization shall occur during system startup.

returns the value returned is either null or it resides in immortal memory.

```
@SCJAllowed
@SCJMaySelfSuspend(false)
@SCJMayAllocate({})
@SCJPhase({
    javax.safetycritical.annotate.Phase.STARTUP,
    javax.safetycritical.annotate.Phase.INITIALIZATION,
    javax.safetycritical.annotate.Phase.RUN,
    javax.safetycritical.annotate.Phase.CLEANUP })
public static int identityHashCode(Object x)
```

Does not allow argument ”x” to escape local variables. Allocates no memory.

```
@SCJAllowed
@SCJMaySelfSuspend(false)
@SCJMayAllocate({})
@SCJPhase({
    javax.safetycritical.annotate.Phase.STARTUP,
    javax.safetycritical.annotate.Phase.INITIALIZATION,
    javax.safetycritical.annotate.Phase.RUN,
    javax.safetycritical.annotate.Phase.CLEANUP })
public static long nanoTime( )
```

Allocates no memory.

B.3.44 CLASS Thread

```
@SCJAllowed
public class Thread implements java.lang.Runnable extends java.lang.Object
```

The Thread class is not directly available to the application in SCJ. However, some of its static methods are used, and the infrastructure will extend from this class and hence some of its methods are inherited.

Methods

```
@SCJAllowed(javax.safetycritical.annotate.Level.LEVEL_2)
@SCJPhase({
   javax.safetycritical.annotate.Phase.INITIALIZATION,
   javax.safetycritical.annotate.Phase.RUN,
   javax.safetycritical.annotate.Phase.CLEANUP})
@SCJMaySelfSuspend(false)
@SCJMayAllocate({})
public static java.lang.Thread.UncaughtExceptionHandler
     getDefaultUncaughtExceptionHandler( )
```

Gets the current thread's default uncaught exeption handler. Allocates no memory. Does not allow this to escape local variables. The result returned from this method may reside in scoped memory in some scope that encloses this.

returns the default handler for uncaught exceptions.

```
@SCJAllowed(javax.safetycritical.annotate.Level.LEVEL_2)
@SCJPhase({
   javax.safetycritical.annotate.Phase.INITIALIZATION,
   javax.safetycritical.annotate.Phase.RUN,
   javax.safetycritical.annotate.Phase.CLEANUP})
@SCJMaySelfSuspend(false)
@SCJMayAllocate({})
public java.lang.Thread.UncaughtExceptionHandler getUncaughtExceptionHandler( )
```

Get the thread's uncaught exception handler.

returns the handler invoked when this thread abruptly terminates due to an uncaught exception. Allocates no memory. Does not allow "this" to escape local variables. The result returned from this method may reside in scoped memory in some scope that encloses "this".

```
@SCJAllowed(javax.safetycritical.annotate.Level.LEVEL_2)
@SCJPhase({
   javax.safetycritical.annotate.Phase.INITIALIZATION,
   javax.safetycritical.annotate.Phase.RUN,
   javax.safetycritical.annotate.Phase.CLEANUP})
@SCJMaySelfSuspend(false)
@SCJMayAllocate({})
public void interrupt( )
```

Interrupts the thread. Allocates no memory. Does not allow this to escape local variables.

```
@SCJAllowed(javax.safetycritical.annotate.Level.LEVEL_2)
@SCJMaySelfSuspend(false)
@SCJPhase({
   javax.safetycritical.annotate.Phase.INITIALIZATION,
   javax.safetycritical.annotate.Phase.RUN,
   javax.safetycritical.annotate.Phase.CLEANUP})
@SCJMayAllocate({})
public static boolean interrupted( )
```

Tests whether the thread has been interrupted. The interrupted status of the thread is cleared by this method. Allocates no memory. Does not allow this to escape local variables.

returns true if the current thread has been interrupted.

```
@SCJAllowed(javax.safetycritical.annotate.Level.LEVEL_2)
@SCJPhase({
   javax.safetycritical.annotate.Phase.INITIALIZATION,
   javax.safetycritical.annotate.Phase.RUN,
   javax.safetycritical.annotate.Phase.CLEANUP})
@SCJMaySelfSuspend(false)
@SCJMayAllocate({})
public final boolean isAlive( )
```

Tests whether the thread is alive.

returns true if the current thread has not returned from run(). Allocates no memory. Does not allow this to escape local variables.

```
@SCJAllowed(javax.safetycritical.annotate.Level.LEVEL_2)
@SCJPhase({
   javax.safetycritical.annotate.Phase.INITIALIZATION,
   javax.safetycritical.annotate.Phase.RUN,
   javax.safetycritical.annotate.Phase.CLEANUP})
@SCJMaySelfSuspend(false)
@SCJMayAllocate({})
public boolean isInterrupted( )
```

Tests whether the thread has been interrupted. The interrupted status of the thread is not affected by this method. Allocates no memory. Does not allow this to escape local variables.

returns true if a thread has been interrupted.

```
@SCJAllowed(javax.safetycritical.annotate.Level.SUPPORT)
@SCJMayAllocate({
   javax.safetycritical.annotate.AllocationContext.CURRENT,
   javax.safetycritical.annotate.AllocationContext.INNER,
   javax.safetycritical.annotate.AllocationContext.OUTER})
@SCJMaySelfSuspend(true)
@SCJPhase({
   javax.safetycritical.annotate.Phase.STARTUP,
   javax.safetycritical.annotate.Phase.INITIALIZATION,
   javax.safetycritical.annotate.Phase.RUN,
   javax.safetycritical.annotate.Phase.CLEANUP })
public void run( )
```

This method is overridden by the application to do the work desired for this thread. This method should not be directly called by the application.

```
@SCJPhase({
   javax.safetycritical.annotate.Phase.INITIALIZATION,
   javax.safetycritical.annotate.Phase.RUN,
   javax.safetycritical.annotate.Phase.CLEANUP})
@SCJMaySelfSuspend(false)
@SCJMayAllocate({javax.safetycritical.annotate.AllocationContext.CURRENT})
@SCJAllowed
public java.lang.String toString( )
```

Gets the name and priority for the thread.

returns a string representation of this thread, including the thread's name and priority. Does not allow this to escape local variables. Allocates a String and associated internal "structure" (e.g. char[]) in caller's scope.

```
@SCJAllowed(javax.safetycritical.annotate.Level.LEVEL_2)
@SCJPhase({
   javax.safetycritical.annotate.Phase.INITIALIZATION,
   javax.safetycritical.annotate.Phase.RUN,
   javax.safetycritical.annotate.Phase.CLEANUP})
@SCJMaySelfSuspend(false)
@SCJMayAllocate({})
public static void yield( )
```

Causes the thread to yield to other threads that may be ready to run. Causes the currently executing thread object to temporary pause and allow other threads to execute.

B.3.45 CLASS **Throwable**

```
@SCJAllowed
public class Throwable implements java.io.Serializable extends
    java.lang.Object
```

The Throwable class is the superclass of all errors and exceptions in the Java language. Only objects that are instances of this class (or one of its subclasses) are thrown by the Java Virtual Machine or can be thrown by the Java throw statement. Similarly, only this class or one of its subclasses can be the argument type in a catch clause. For the purposes of compile-time checking of exceptions, Throwable and any subclass of Throwable that is not also a subclass of either RuntimeException or Error are regarded as checked exceptions.

Constructors

```
@SCJAllowed
@SCJMaySelfSuspend(false)
@SCJMayAllocate({javax.safetycritical.annotate.AllocationContext.CURRENT})
@SCJPhase({
   javax.safetycritical.annotate.Phase.STARTUP,
   javax.safetycritical.annotate.Phase.INITIALIZATION,
   javax.safetycritical.annotate.Phase.RUN,
   javax.safetycritical.annotate.Phase.CLEANUP })
public Throwable(String msg, Throwable cause)
```

Constructs a Throwable object with an optional detail message and an optional cause. If cause is null, Services.captureStackBacktrace(this) is called to save the backtrace associated with the current thread. If cause is not null, Services.captureStackBacktrace(this) is not called to avoid overwriting the backtrace associated with the cause.

msg — the detail message for this Throwable object.

cause — the cause of this exception.

```
@SCJAllowed
@SCJMaySelfSuspend(false)
@SCJMayAllocate({
   javax.safetycritical.annotate.AllocationContext.CURRENT,
   javax.safetycritical.annotate.AllocationContext.INNER,
   javax.safetycritical.annotate.AllocationContext.OUTER})
@SCJPhase({
   javax.safetycritical.annotate.Phase.STARTUP,
   javax.safetycritical.annotate.Phase.INITIALIZATION,
   javax.safetycritical.annotate.Phase.RUN,
   javax.safetycritical.annotate.Phase.CLEANUP })
public Throwable( )
```

This constructor behaves the same as calling Throwable(String, Throwable) with the arguments (null, null).

```
@SCJAllowed
@SCJMaySelfSuspend(false)
@SCJMayAllocate({javax.safetycritical.annotate.AllocationContext.CURRENT})
@SCJPhase({
   javax.safetycritical.annotate.Phase.STARTUP,
   javax.safetycritical.annotate.Phase.INITIALIZATION,
   javax.safetycritical.annotate.Phase.RUN,
   javax.safetycritical.annotate.Phase.CLEANUP })
public Throwable(Throwable cause)
```

This constructor behaves the same as calling Throwable(String, Throwable) with the arguments (null, cause).

```
@SCJAllowed
@SCJMaySelfSuspend(false)
@SCJMayAllocate({javax.safetycritical.annotate.AllocationContext.CURRENT})
@SCJPhase({
   javax.safetycritical.annotate.Phase.STARTUP,
   javax.safetycritical.annotate.Phase.INITIALIZATION,
   javax.safetycritical.annotate.Phase.RUN,
   javax.safetycritical.annotate.Phase.CLEANUP })
public Throwable(String msg)
```

This constructor behaves the same as calling Throwable(String, Throwable) with the arguments (msg, null).

Methods

```
@SCJAllowed
@SCJMaySelfSuspend(false)
@SCJMayAllocate({})
@SCJPhase({
   javax.safetycritical.annotate.Phase.STARTUP,
   javax.safetycritical.annotate.Phase.INITIALIZATION,
   javax.safetycritical.annotate.Phase.RUN,
   javax.safetycritical.annotate.Phase.CLEANUP })
public java.lang.Throwable getCause( )
```

returns a reference to the cause that was supplied as an argument to the constructor, or null if no cause was specified at construction time. Performs no memory allocation.

```
@SCJAllowed
@SCJMaySelfSuspend(false)
@SCJMayAllocate({})
@SCJPhase({
   javax.safetycritical.annotate.Phase.STARTUP,
   javax.safetycritical.annotate.Phase.INITIALIZATION,
   javax.safetycritical.annotate.Phase.RUN,
   javax.safetycritical.annotate.Phase.CLEANUP })
public java.lang.String getMessage( )
```

returns a reference to the message that was supplied as an argument to the constructor, or null if no message was specified at construction time. Performs no memory allocation.

```
@SCJAllowed
@SCJMaySelfSuspend(false)
@SCJMayAllocate({javax.safetycritical.annotate.AllocationContext.CURRENT})
@SCJPhase({
   javax.safetycritical.annotate.Phase.STARTUP,
   javax.safetycritical.annotate.Phase.INITIALIZATION,
   javax.safetycritical.annotate.Phase.RUN,
   javax.safetycritical.annotate.Phase.CLEANUP })
public java.lang.StackTraceElement[] getStackTrace( )
```

Allocates a StackTraceElement array, StackTraceElement objects, and all internal structure, including String objects referenced from each StackTraceElement to represent the stack backtrace information available for the exception that was most recently associated with this Throwable object.

Each Schedulable maintains a single buffer to contain the stack backtrace information associated with the most recent invocation of System.captureStackBacktrace. The size of this buffer is specified by providing a SchedulableSizingParameters object as an argument to construction of the Schedulable. Most commonly, Services.captureStackBacktrace is invoked from within the constructor of java.lang.Throwable . getStackTrace returns the contents of this single backtrace buffer information.

If Services.captureStackBacktrace has been invoked within this thread more recently than the construction of this Throwable, then the stack trace information returned from this method may not represent the stack backtrace for this particular Throwable.

```
@SCJMayAllocate({javax.safetycritical.annotate.AllocationContext.CURRENT})
@SCJAllowed
@SCJMaySelfSuspend(false)
@SCJPhase({
```

```
    javax.safetycritical.annotate.Phase.STARTUP,
    javax.safetycritical.annotate.Phase.INITIALIZATION,
    javax.safetycritical.annotate.Phase.RUN,
    javax.safetycritical.annotate.Phase.CLEANUP })
public void printStackTrace(PrintStream stream)
```

Print the stack trace of this Throwable to the given stream.

The printed stack trace contains the result of toString() as the first line followed by one line for each stack trace element that contains the name of the method or constructor, optionally followed by the source file name and source file line number when available.

stream — the stream to print to.

```
@SCJMayAllocate({javax.safetycritical.annotate.AllocationContext.CURRENT})
@SCJAllowed
@SCJMaySelfSuspend(false)
@SCJPhase({
    javax.safetycritical.annotate.Phase.STARTUP,
    javax.safetycritical.annotate.Phase.INITIALIZATION,
    javax.safetycritical.annotate.Phase.RUN,
    javax.safetycritical.annotate.Phase.CLEANUP })
public void printStackTrace( )
```

B.3.46 CLASS UnsatisfiedLinkError

```
@SCJAllowed
public class UnsatisfiedLinkError implements java.io.Serializable extends
      java.lang.RuntimeException
```

Thrown if the Java Virtual Machine cannot find an appropriate native-language definition of a method declared native.

Constructors

```
@SCJMayAllocate({javax.safetycritical.annotate.AllocationContext.CURRENT})
@SCJAllowed
@SCJMaySelfSuspend(false)
@SCJPhase({
    javax.safetycritical.annotate.Phase.STARTUP,
    javax.safetycritical.annotate.Phase.INITIALIZATION,
    javax.safetycritical.annotate.Phase.RUN,
    javax.safetycritical.annotate.Phase.CLEANUP })
public UnsatisfiedLinkError( )
```

Shall not copy ”this” to any instance or static field.

Allocates an application- and implementation-dependent amount of memory in the current scope (to represent stack backtrace).

```
@SCJMayAllocate({javax.safetycritical.annotate.AllocationContext.CURRENT})
@SCJAllowed
@SCJMaySelfSuspend(false)
@SCJPhase({
   javax.safetycritical.annotate.Phase.STARTUP,
   javax.safetycritical.annotate.Phase.INITIALIZATION,
   javax.safetycritical.annotate.Phase.RUN,
   javax.safetycritical.annotate.Phase.CLEANUP })
public UnsatisfiedLinkError(String msg)
```

Shall not copy ”this” to any instance or static field. The scope containing the msg argument must enclose the scope containing ”this”. Otherwise, an IllegalAssignmentError will be thrown.

Allocates an application- and implementation-dependent amount of memory in the current scope (to represent stack backtrace).

B.3.47 CLASS UnsupportedOperationException

```
@SCJAllowed
public class UnsupportedOperationException implements java.io.Serializable
    extends java.lang.RuntimeException
```

Thrown to indicate that the requested operation is not supported.

Constructors

```
@SCJAllowed
@SCJMaySelfSuspend(false)
@SCJMayAllocate({javax.safetycritical.annotate.AllocationContext.CURRENT})
@SCJPhase({
   javax.safetycritical.annotate.Phase.STARTUP,
   javax.safetycritical.annotate.Phase.INITIALIZATION,
   javax.safetycritical.annotate.Phase.RUN,
   javax.safetycritical.annotate.Phase.CLEANUP })
public UnsupportedOperationException( )
```

Shall not copy ”this” to any instance or static field.

Allocates an application- and implementation-dependent amount of memory in the current scope (to represent stack backtrace).

```
@SCJAllowed
@SCJMaySelfSuspend(false)
@SCJMayAllocate({javax.safetycritical.annotate.AllocationContext.CURRENT})
@SCJPhase({
   javax.safetycritical.annotate.Phase.STARTUP,
   javax.safetycritical.annotate.Phase.INITIALIZATION,
   javax.safetycritical.annotate.Phase.RUN,
   javax.safetycritical.annotate.Phase.CLEANUP })
public UnsupportedOperationException(String msg)
```

Shall not copy ”this” to any instance or static field. The scope containing the msg argument must enclose the scope containing ”this”. Otherwise, an IllegalAssignmentError will be thrown.

Allocates an application- and implementation-dependent amount of memory in the current scope (to represent stack backtrace).

```
@SCJAllowed
@SCJMaySelfSuspend(false)
@SCJMayAllocate({javax.safetycritical.annotate.AllocationContext.CURRENT})
@SCJPhase({
   javax.safetycritical.annotate.Phase.STARTUP,
   javax.safetycritical.annotate.Phase.INITIALIZATION,
   javax.safetycritical.annotate.Phase.RUN,
   javax.safetycritical.annotate.Phase.CLEANUP })
public UnsupportedOperationException(String msg, Throwable t)
```

Shall not copy ”this” to any instance or static field. The scope containing the msg argument must enclose the scope containing ”this”. Otherwise, an IllegalAssignmentError will be thrown.

Allocates an application- and implementation-dependent amount of memory in the current scope (to represent stack backtrace).

```
@SCJAllowed
@SCJMaySelfSuspend(false)
@SCJMayAllocate({javax.safetycritical.annotate.AllocationContext.CURRENT})
@SCJPhase({
   javax.safetycritical.annotate.Phase.STARTUP,
   javax.safetycritical.annotate.Phase.INITIALIZATION,
   javax.safetycritical.annotate.Phase.RUN,
   javax.safetycritical.annotate.Phase.CLEANUP })
public UnsupportedOperationException(Throwable t)
```

Shall not copy ”this” to any instance or static field. The scope containing the msg argument must enclose the scope containing ”this”. Otherwise, an IllegalAssignmentError will be thrown.

Allocates an application- and implementation-dependent amount of memory in the current scope (to represent stack backtrace).

B.3.48 CLASS VirtualMachineError

```
@SCJAllowed
public class VirtualMachineError implements java.io.Serializable extends
    java.lang.Error
```

Thrown to indicate that the Java Virtual Machine is broken or has run out of resources necessary for it to continue operating.

Constructors

```
@SCJAllowed
@SCJMaySelfSuspend(false)
@SCJMayAllocate({
   javax.safetycritical.annotate.AllocationContext.CURRENT,
   javax.safetycritical.annotate.AllocationContext.INNER,
   javax.safetycritical.annotate.AllocationContext.OUTER})
@SCJPhase({
   javax.safetycritical.annotate.Phase.STARTUP,
   javax.safetycritical.annotate.Phase.INITIALIZATION,
   javax.safetycritical.annotate.Phase.RUN,
   javax.safetycritical.annotate.Phase.CLEANUP })
public VirtualMachineError( )
```

Shall not copy ”this” to any instance or static field.

Allocates an application- and implementation-dependent amount of memory in the current scope (to represent stack backtrace).

```
@SCJAllowed
@SCJMaySelfSuspend(false)
@SCJMayAllocate({
   javax.safetycritical.annotate.AllocationContext.CURRENT,
   javax.safetycritical.annotate.AllocationContext.INNER,
   javax.safetycritical.annotate.AllocationContext.OUTER})
@SCJPhase({
   javax.safetycritical.annotate.Phase.STARTUP,
   javax.safetycritical.annotate.Phase.INITIALIZATION,
   javax.safetycritical.annotate.Phase.RUN,
   javax.safetycritical.annotate.Phase.CLEANUP })
public VirtualMachineError(String msg)
```

Shall not copy ”this” to any instance or static field. The scope containing the msg argument must enclose the scope containing ”this”. Otherwise, an IllegalAssignmentError will be thrown.

Allocates an application- and implementation-dependent amount of memory in the current scope (to represent stack backtrace).

B.3.49 CLASS **Void**

@SCJAllowed
public final class Void extends java.lang.Object

The Void class is an uninstantiable placeholder class to hold a reference to the Class object representing the Java keyword void.

Fields

@SCJAllowed
public static final java.lang.**Class**<java.lang.**Void**> TYPE

Appendix C

Javadoc Description of Package javax.microedition.io

Package Contents *Page*

C.1 Classes

C.2 Interfaces

C.2.1 INTERFACE Connection

```
@SCJAllowed
public interface Connection
```

This is the most basic type of generic connection. Only the close method is defined. No open method is defined here because opening is always done using the Connector.open() methods.

Methods

```
@SCJAllowed
@SCJMaySelfSuspend(true)
@SCJMayAllocate({
   javax.safetycritical.annotate.AllocationContext.CURRENT,
   javax.safetycritical.annotate.AllocationContext.INNER,
   javax.safetycritical.annotate.AllocationContext.OUTER})
@SCJPhase({
   javax.safetycritical.annotate.Phase.STARTUP,
   javax.safetycritical.annotate.Phase.INITIALIZATION,
   javax.safetycritical.annotate.Phase.RUN,
   javax.safetycritical.annotate.Phase.CLEANUP })
public void close( )
```

Close the connection.

When a connection has been closed, access to any of its methods that involve an I/O operation will cause an IOException to be thrown. Closing an already closed connection has no effect. Streams derived from the connection may be open when the method is called. Any open streams will cause the connection to be held open until they themselves are closed. In this latter case access to the open streams is permitted, but access to the connection is not.

Throws IOException if an I/O error occurs

Throws IllegalArgumentException

Throws ConnectionNotFoundException

C.2.2 INTERFACE **InputConnection**

```
@SCJAllowed
public interface InputConnection extends javax.microedition.io.Connection
```

This interface defines the capabilities that an input stream connection must have.

Methods

```
@SCJAllowed
@SCJMayAllocate({
   javax.safetycritical.annotate.AllocationContext.CURRENT,
   javax.safetycritical.annotate.AllocationContext.INNER,
   javax.safetycritical.annotate.AllocationContext.OUTER})
@SCJMaySelfSuspend(true)
@SCJPhase({
   javax.safetycritical.annotate.Phase.STARTUP,
   javax.safetycritical.annotate.Phase.INITIALIZATION,
   javax.safetycritical.annotate.Phase.RUN,
   javax.safetycritical.annotate.Phase.CLEANUP })
public java.io.DataInputStream openDataInputStream( )
```

Open and return a data input stream for a connection.

returns An input stream.

Throws IOException if an I/O error occurs.

```
@SCJAllowed
@SCJMaySelfSuspend(true)
@SCJMayAllocate({
   javax.safetycritical.annotate.AllocationContext.CURRENT,
   javax.safetycritical.annotate.AllocationContext.INNER,
   javax.safetycritical.annotate.AllocationContext.OUTER})
@SCJPhase({
   javax.safetycritical.annotate.Phase.STARTUP,
   javax.safetycritical.annotate.Phase.INITIALIZATION,
   javax.safetycritical.annotate.Phase.RUN,
   javax.safetycritical.annotate.Phase.CLEANUP })
public java.io.InputStream openInputStream( )
```

Open and return an input stream for a connection.

returns An input stream.

Throws IOException if an I/O error occurs.

C.2.3 INTERFACE OutputConnection

```
@SCJAllowed
public interface OutputConnection extends javax.microedition.io.Connection
```

This interface defines the capabilities that an output stream connection must have.

Methods

```
@SCJAllowed
@SCJMayAllocate({
    javax.safetycritical.annotate.AllocationContext.CURRENT,
    javax.safetycritical.annotate.AllocationContext.INNER,
    javax.safetycritical.annotate.AllocationContext.OUTER})
@SCJMaySelfSuspend(true)
@SCJPhase({
    javax.safetycritical.annotate.Phase.STARTUP,
    javax.safetycritical.annotate.Phase.INITIALIZATION,
    javax.safetycritical.annotate.Phase.RUN,
    javax.safetycritical.annotate.Phase.CLEANUP })
public java.io.DataOutputStream openDataOutputStream( )
```

Open and return a data output stream for a connection.

returns An output stream.

Throws IOException if an I/O error occurs.

```
@SCJAllowed
@SCJMaySelfSuspend(true)
@SCJMayAllocate({
    javax.safetycritical.annotate.AllocationContext.CURRENT,
    javax.safetycritical.annotate.AllocationContext.INNER,
    javax.safetycritical.annotate.AllocationContext.OUTER})
@SCJPhase({
    javax.safetycritical.annotate.Phase.STARTUP,
    javax.safetycritical.annotate.Phase.INITIALIZATION,
    javax.safetycritical.annotate.Phase.RUN,
    javax.safetycritical.annotate.Phase.CLEANUP })
public java.io.OutputStream openOutputStream( )
```

Open and return an output stream for a connection.

returns An output stream.

Throws IOException if an I/O error occurs.

C.2.4 INTERFACE StreamConnection

@SCJAllowed
public interface StreamConnection **extends** javax.microedition.io.InputConnection,
javax.microedition.io.OutputConnection

This interface defines the capabilities that a stream connection must have.

In a typical implementation of this interface, all StreamConnections have one underlying InputStream and one OutputStream. Opening a DataInputStream counts as opening an InputStream and opening a DataOutputStream counts as opening an OutputStream. Trying to open another InputStream or OutputStream causes an IOException. Trying to open the InputStream or OutputStream after they have been closed causes an IOException.

The methods of StreamConnection are not synchronized. The only stream method that can be called safely in another thread is close.

C.3 Classes

C.3.1 CLASS ConnectionNotFoundException

@SCJAllowed
public class ConnectionNotFoundException **extends** java.io.IOException

This class is used to signal that a connection target cannot be found, or the protocol type is not supported.

Constructors

```
@SCJAllowed
@SCJMayAllocate({})
@SCJMaySelfSuspend(false)
@SCJPhase({
    javax.safetycritical.annotate.Phase.STARTUP,
    javax.safetycritical.annotate.Phase.INITIALIZATION,
    javax.safetycritical.annotate.Phase.RUN,
    javax.safetycritical.annotate.Phase.CLEANUP })
public ConnectionNotFoundException(String s)
```

Constructs a ConnectionNotFoundException with the specified detail message. A detail message is a String that describes this particular exception.

s — the detail message. If s is null, no detail message is provided.

```
@SCJAllowed
@SCJMayAllocate({})
@SCJMaySelfSuspend(false)
@SCJPhase({
   javax.safetycritical.annotate.Phase.STARTUP,
   javax.safetycritical.annotate.Phase.INITIALIZATION,
   javax.safetycritical.annotate.Phase.RUN,
   javax.safetycritical.annotate.Phase.CLEANUP })
public ConnectionNotFoundException( )
```

This constructor behaves the same as calling ConnectionNotFoundException(String) with the arguments (null).

C.3.2 CLASS Connector

```
@SCJAllowed
public class Connector extends java.lang.Object
```

This class is a factory for use by applications to dynamically create Connection objects. The application provides a specified name that this factory will use to identify an appropriate connection to a device or interface. The specified name conforms to the URL format defined in RFC 2396. The specified name uses this format:

{scheme}:[{target}][{params}]

where {scheme} is the name of a protocol such as *http* .

The {target} is normally some kind of network address or other interface such as a file designation.

Any {params} are formed as a series of equates of the form ";x=y". Example: ";type=a".

Within this format, the application may provide an optional second parameter to the open function. This second parameter is a mode flag to indicate the intentions of the calling code to the protocol handler. The options here specify whether the connection will be used to read (READ), write (WRITE), or both (READ_WRITE). Each protocol specifies which flag settings are permitted. For example, a printer would likely not permit read access, so it might throw an IllegalArgumentException. If not specified, READ_WRITE mode is used by default. // *

// * In addition, a third parameter may be specified as a boolean flag // * indicating that the application intends to handle timeout exceptions. // * If this flag is true, the protocol implementation may throw an // * InterruptedIOException if a timeout condition is detected. // * This flag may be ignored by the protocol

handler; the // * InterruptedIOException may not actually be thrown. // * If this parameter is false, the protocol shall not throw // * the InterruptedIOException.

Fields

```
@SCJAllowed
public static final int READ
```

Access mode READ.

```
@SCJAllowed
public static final int READ_WRITE
```

Access mode READ_WRITE.

```
@SCJAllowed
public static final int WRITE
```

Access mode WRITE.

Methods

```
@SCJAllowed
@SCJMaySelfSuspend(true)
@SCJMayAllocate({
    javax.safetycritical.annotate.AllocationContext.CURRENT,
    javax.safetycritical.annotate.AllocationContext.INNER,
    javax.safetycritical.annotate.AllocationContext.OUTER})
@SCJPhase({
    javax.safetycritical.annotate.Phase.STARTUP,
    javax.safetycritical.annotate.Phase.INITIALIZATION,
    javax.safetycritical.annotate.Phase.RUN,
    javax.safetycritical.annotate.Phase.CLEANUP })
public static javax.microedition.io.Connection open(String name)
  throws java.io.IOException, java.lang.SecurityException
```

Create and open a Connection. This method is the same as calling open(name, READ_WRITE).

name — The URL for the connection.

returns a new Connection object.

Throws IllegalArgumentException if a parameter is invalid.

Throws ConnectionNotFoundException if the target of the name cannot be found, or if the requested protocol type is not supported.

Throws IOException if some other kind of I/O error occurs.

Throws SecurityException if access to the protocol handler is prohibited.

```
@SCJAllowed
@SCJMaySelfSuspend(true)
@SCJMayAllocate({
    javax.safetycritical.annotate.AllocationContext.CURRENT,
    javax.safetycritical.annotate.AllocationContext.INNER,
    javax.safetycritical.annotate.AllocationContext.OUTER})
@SCJPhase({
    javax.safetycritical.annotate.Phase.STARTUP,
    javax.safetycritical.annotate.Phase.INITIALIZATION,
    javax.safetycritical.annotate.Phase.RUN,
    javax.safetycritical.annotate.Phase.CLEANUP })
public static javax.microedition.io.Connection open(String name, int mode)
  throws java.io.IOException, java.lang.SecurityException
```

Create and open a Connection with a specified name and access mode.

name — The URL for the connection.

mode — The access mode (i.e., READ, WRITE, or READ_WRITE.)

returns A new Connection object.

Throws IllegalArgumentException if a parameter is invalid.

Throws ConnectionNotFoundException if the target of the name cannot be found, or if the requested protocol type is not supported.

Throws IOException if some other kind of I/O error occurs.

Throws SecurityException if access to the protocol handler is prohibited.

```
@SCJAllowed
@SCJMaySelfSuspend(true)
@SCJMayAllocate({
    javax.safetycritical.annotate.AllocationContext.CURRENT,
    javax.safetycritical.annotate.AllocationContext.INNER,
    javax.safetycritical.annotate.AllocationContext.OUTER})
@SCJPhase({
    javax.safetycritical.annotate.Phase.STARTUP,
    javax.safetycritical.annotate.Phase.INITIALIZATION,
    javax.safetycritical.annotate.Phase.RUN,
    javax.safetycritical.annotate.Phase.CLEANUP })
public static java.io.DataInputStream openDataInputStream(String name)
  throws java.io.IOException, java.lang.SecurityException
```

Create and open a connection input stream.

name — The URL for the connection.

returns A DataInputStream.

Throws IllegalArgumentException if a parameter is invalid.

Throws ConnectionNotFoundException if the target of the name cannot be found, or if the requested protocol type is not supported.

Throws IOException if some other kind of I/O error occurs.

Throws SecurityException if access to the protocol handler is prohibited.

```
@SCJAllowed
@SCJMaySelfSuspend(true)
@SCJMayAllocate({
    javax.safetycritical.annotate.AllocationContext.CURRENT,
    javax.safetycritical.annotate.AllocationContext.INNER,
    javax.safetycritical.annotate.AllocationContext.OUTER})
@SCJPhase({
    javax.safetycritical.annotate.Phase.STARTUP,
    javax.safetycritical.annotate.Phase.INITIALIZATION,
    javax.safetycritical.annotate.Phase.RUN,
    javax.safetycritical.annotate.Phase.CLEANUP })
public static java.io.DataOutputStream openDataOutputStream(String name)
  throws java.io.IOException, java.lang.SecurityException
```

Create and open a connection output stream.

name — The URL for the connection.

returns A DataOutputStream.

Throws IllegalArgumentException if a parameter is invalid.

Throws ConnectionNotFoundException if the target of the name cannot be found, or if the requested protocol type is not supported.

Throws IOException if some other kind of I/O error occurs.

Throws SecurityException if access to the protocol handler is prohibited.

```
@SCJAllowed
@SCJMaySelfSuspend(true)
@SCJMayAllocate({
    javax.safetycritical.annotate.AllocationContext.CURRENT,
    javax.safetycritical.annotate.AllocationContext.INNER,
    javax.safetycritical.annotate.AllocationContext.OUTER})
@SCJPhase({
    javax.safetycritical.annotate.Phase.STARTUP,
```

```
    javax.safetycritical.annotate.Phase.INITIALIZATION,
    javax.safetycritical.annotate.Phase.RUN,
    javax.safetycritical.annotate.Phase.CLEANUP })
public static java.io.InputStream openInputStream(String name)
  throws java.io.IOException, java.lang.SecurityException
```

Create and open a connection input stream.

name — The URL for the connection.

returns An InputStream.

Throws IllegalArgumentException if a parameter is invalid.

Throws ConnectionNotFoundException if the target of the name cannot be found, or if the requested protocol type is not supported.

Throws IOException if some other kind of I/O error occurs.

Throws SecurityException if access to the protocol handler is prohibited.

```
@SCJAllowed
@SCJMaySelfSuspend(true)
@SCJMayAllocate({
    javax.safetycritical.annotate.AllocationContext.CURRENT,
    javax.safetycritical.annotate.AllocationContext.INNER,
    javax.safetycritical.annotate.AllocationContext.OUTER})
@SCJPhase({
    javax.safetycritical.annotate.Phase.STARTUP,
    javax.safetycritical.annotate.Phase.INITIALIZATION,
    javax.safetycritical.annotate.Phase.RUN,
    javax.safetycritical.annotate.Phase.CLEANUP })
public static java.io.OutputStream openOutputStream(String name)
  throws java.io.IOException, java.lang.SecurityException
```

Create and open a connection output stream.

name — The URL for the connection.

returns An OutputStream.

Throws IllegalArgumentException if a parameter is invalid.

Throws ConnectionNotFoundException if the target of the name cannot be found, or if the requested protocol type is not supported.

Throws IOException if some other kind of I/O error occurs.

Throws SecurityException if access to the protocol handler is prohibited.

Appendix D

Javadoc Description of Package javax.realtime

Package Contents *Page*

Classes

D.1 Classes

D.2 Interfaces

D.2.1 INTERFACE AsyncTimable

@SCJAllowed
public interface AsyncTimable **extends** javax.realtime.Timable

An interface to indicate it they can be associated with a Clock and be suspended waiting for time events based on that clock.

D.2.2 INTERFACE BoundAsyncBaseEventHandler

@SCJAllowed
public interface BoundAsyncBaseEventHandler **extends**
javax.realtime.BoundSchedulable

An empty interface. It is required in order to allow references to all bound handlers.

D.2.3 INTERFACE BoundRealtimeExecutor

@SCJAllowed
public interface BoundRealtimeExecutor

This interface denotes all RTSJ and SCJ objects that encapsulate execution. In SCJ, this type includes Schedulable and InterruptServiceRoutine objects. It is used by Affinity to remove the need to have a reference into the javax.realtime.device package.

Methods

```
@SCJMaySelfSuspend(false)
@SCJPhase({
   javax.safetycritical.annotate.Phase.STARTUP,
   javax.safetycritical.annotate.Phase.INITIALIZATION,
   javax.safetycritical.annotate.Phase.RUN,
   javax.safetycritical.annotate.Phase.CLEANUP})
@SCJMayAllocate({})
@SCJAllowed(javax.safetycritical.annotate.Level.LEVEL_1)
public javax.realtime.Affinity getAffinity( )
```

Determine the affinity set instance associated with {@code task}.

returns The associated affinity.

```
@SCJMaySelfSuspend(false)
@SCJPhase({
   javax.safetycritical.annotate.Phase.STARTUP,
   javax.safetycritical.annotate.Phase.INITIALIZATION})
@SCJMayAllocate({})
@SCJAllowed(javax.safetycritical.annotate.Level.LEVEL_1)
public void setAffinity(Affinity set)
  throws java.lang.IllegalArgumentException,
     javax.realtime.ProcessorAffinityException, java.lang.NullPointerException
```

Set the processor affinity of a {@code task} to {@code set} with immediate effect.

set — is the processor affinity

Throws IllegalArgumentException when the intersection of {@code set} the affinity of any {@code ThreadGroup} instance containing {@code task} is empty.

Throws ProcessorAffinityException is thrown when the runtime fails to set the affinity for platform-specific reasons.

Throws NullPointerException when {@code set} is {@code null}.

D.2.4 INTERFACE **BoundSchedulable**

```
@SCJAllowed
public interface BoundSchedulable extends javax.realtime.Schedulable,
     javax.realtime.BoundRealtimeExecutor
```

A marker interface to provide a type safe reference to all schedulables that are bound to a single underlying thread.

D.2.5 INTERFACE **Chronograph**

```
@SCJAllowed
public interface Chronograph
```

The interface for all devices that support the measurement of time. All Chronograph implementations use time values derived from HighResolutionTime, which expresses its time in milliseconds and nanoseconds. However, for an application-defined clock, its time values are not necessarily related to the wall clock time

in any particular fashion. For instance, they could represent a count of wheel revolutions or particular event detections. In any case, the time values for every clock shall be mapped to milliseconds and nanoseconds in a manner that is computationally appropriate.

Methods

```
@SCJAllowed
@SCJMaySelfSuspend(false)
@SCJMayAllocate({javax.safetycritical.annotate.AllocationContext.CURRENT})
@SCJPhase({
   javax.safetycritical.annotate.Phase.STARTUP,
   javax.safetycritical.annotate.Phase.INITIALIZATION,
   javax.safetycritical.annotate.Phase.RUN,
   javax.safetycritical.annotate.Phase.CLEANUP })
public javax.realtime.RelativeTime getEpochOffset( )
```

Determines the time on the real-time clock when this chronograph was zero.

returns A newly allocated RelativeTime object with the real-time clock as its chronograph and containing the time from the real-time clock when this chronograph was zero.

Throws UnsupportedOperationException when this chronograph does not have the concept of an Epoch.

```
@SCJAllowed
@SCJMaySelfSuspend(false)
@SCJMayAllocate({javax.safetycritical.annotate.AllocationContext.CURRENT})
@SCJPhase({
   javax.safetycritical.annotate.Phase.STARTUP,
   javax.safetycritical.annotate.Phase.INITIALIZATION,
   javax.safetycritical.annotate.Phase.RUN,
   javax.safetycritical.annotate.Phase.CLEANUP })
public javax.realtime.RelativeTime getQueryPrecision(RelativeTime dest)
```

Gets the precision of the time read, defined as the nominal interval between ticks.

dest — is an object that, upon return from this method, shall contain the precision of the time read. If dest is null, this method shall allocate a new RelativeTime instance to hold the returned value.

returns the value of dest if dest is not null, otherwise a new object representing the read precision.

```
@SCJAllowed
@SCJMaySelfSuspend(false)
@SCJMayAllocate({javax.safetycritical.annotate.AllocationContext.CURRENT})
@SCJPhase({
   javax.safetycritical.annotate.Phase.STARTUP,
   javax.safetycritical.annotate.Phase.INITIALIZATION,
   javax.safetycritical.annotate.Phase.RUN,
   javax.safetycritical.annotate.Phase.CLEANUP })
public javax.realtime.RelativeTime getQueryPrecision( )
```

Gets the precision of the time read defined as the nominal interval between ticks. It is the same as calling getQueryPrecision(null).

returns a value representing the read precision.

```
@SCJAllowed
@SCJMaySelfSuspend(false)
@SCJMayAllocate({})
@SCJPhase({
   javax.safetycritical.annotate.Phase.STARTUP,
   javax.safetycritical.annotate.Phase.INITIALIZATION,
   javax.safetycritical.annotate.Phase.RUN,
   javax.safetycritical.annotate.Phase.CLEANUP })
public javax.realtime.AbsoluteTime getTime(AbsoluteTime dest)
```

Gets the current time of this chronograph. The time represented by the returned AbsoluteTime represents some time between the invocation of the method and the return of the method. *Note:* This method will return an absolute time value that represents the chronograph's notion of the current time. For chronographs that do not measure calendar time this absolute time may not represent a wall clock time.

dest — The instance of an AbsoluteTime object which will be updated in place. When dest is not null, the clock association of the dest parameter at the time of the call is ignored; the returned object will be associated with this chronograph. When dest is null, nothing happens.

returns the instance of AbsoluteTime passed as a parameter, representing the current time, associated with this chronograph, or null when dest is null.

```
@SCJAllowed
@SCJMaySelfSuspend(false)
@SCJMayAllocate({javax.safetycritical.annotate.AllocationContext.CURRENT})
@SCJPhase({
   javax.safetycritical.annotate.Phase.STARTUP,
   javax.safetycritical.annotate.Phase.INITIALIZATION,
```

```
    javax.safetycritical.annotate.Phase.RUN,
    javax.safetycritical.annotate.Phase.CLEANUP })
public javax.realtime.AbsoluteTime getTime( )
```

Gets the current time in a newly allocated object. The time represented by the returned AbsoluteTime represents some time between the invocation of the method and the return of the method. *Note:* This method will return an absolute time value that represents the chronograph's notion of the current time. For chronographs that do not measure calendar time this absolute time may not represent a wall clock time.

returns a newly allocated instance of AbsoluteTime in the current allocation context, representing the current time. The returned object is associated with this chronograph.

D.2.6 INTERFACE **Schedulable**

```
@SCJAllowed
public interface Schedulable extends java.lang.Runnable, javax.realtime.Timable
```

In keeping with the RTSJ, SCJ event handlers are schedulable objects. However, the Schedulable interface in the RTSJ is mainly concerned with the getting and setting of the parameter classes. On the contrary, in SCJ, these facilities are not provided. All that is supported in the methods that allow schedulable objects to be interrupted.

Methods

```
@SCJAllowed
@SCJPhase({javax.safetycritical.annotate.Phase.RUN})
@SCJMayAllocate({})
@SCJMaySelfSuspend(false)
public void interrupt( )
```

Behaves as if {@code Thread.interrupt()} were called on the implementation thread underlying this Schedulable. throws IllegalSchedulableStateException when {@code this} is not currently releasable, i.e., its start method has not been called, or it has terminated.

```
@SCJAllowed
@SCJPhase({javax.safetycritical.annotate.Phase.RUN})
@SCJMayAllocate({})
@SCJMaySelfSuspend(false)
public boolean isInterrupted( )
```

Determines whether or not any interrupt is pending.

returns {@code true} when and only when the interrupt is pending.

D.2.7 INTERFACE StaticThrowable

```
@SCJAllowed
public interface StaticThrowable
```

A marker interface to indicate that a Throwable is intended to be created once and reused. Throwables that implement this interface kept their state in a local data structure in the owning schedulable object. This means that data is only valid until the next StaticThrowable is thrown in the that schedulable object. Having a marker interface makes it easier to provide checking tools to ensure the proper throw sequence for all Throwables thrown from application code.

See Also: javax.realtime.ConfigurationParameters

Methods

```
@SCJMayAllocate({})
@SCJAllowed
@SCJMaySelfSuspend(false)
@SCJPhase({
    javax.safetycritical.annotate.Phase.STARTUP,
    javax.safetycritical.annotate.Phase.INITIALIZATION,
    javax.safetycritical.annotate.Phase.RUN,
    javax.safetycritical.annotate.Phase.CLEANUP })
public java.lang.Throwable fillInStackTrace( )
```

Calls the infrastructure to capture the current stack trace in the schedulable object's local memory.

returns a reference to this Throwable.

```
@SCJMayAllocate({})
@SCJAllowed
@SCJMaySelfSuspend(false)
@SCJPhase({
    javax.safetycritical.annotate.Phase.STARTUP,
    javax.safetycritical.annotate.Phase.INITIALIZATION,
    javax.safetycritical.annotate.Phase.RUN,
    javax.safetycritical.annotate.Phase.CLEANUP })
public java.lang.Throwable getCause( )
```

getCause returns the cause of this exception or null when no cause was set by initCause. The cause is another exception that was caught just before this exception was thrown.

returns The cause or null.

```
@SCJMayAllocate({})
@SCJAllowed
@SCJMaySelfSuspend(false)
@SCJPhase({
   javax.safetycritical.annotate.Phase.STARTUP,
   javax.safetycritical.annotate.Phase.INITIALIZATION,
   javax.safetycritical.annotate.Phase.RUN,
   javax.safetycritical.annotate.Phase.CLEANUP })
public java.lang.String getLocalizedMessage( )
```

Subclasses may override this message to get an error message that is localized to the default locale.

By default it returns getMessage().

returns the error message

```
@SCJMayAllocate({})
@SCJAllowed
@SCJMaySelfSuspend(false)
@SCJPhase({
   javax.safetycritical.annotate.Phase.STARTUP,
   javax.safetycritical.annotate.Phase.INITIALIZATION,
   javax.safetycritical.annotate.Phase.RUN,
   javax.safetycritical.annotate.Phase.CLEANUP })
public java.lang.String getMessage( )
```

get the message describing the problem from the schedulable object's local memory.

returns the message given to the constructor or null when no message was set.

```
@SCJMayAllocate({})
@SCJAllowed
@SCJMaySelfSuspend(false)
@SCJPhase({
   javax.safetycritical.annotate.Phase.STARTUP,
   javax.safetycritical.annotate.Phase.INITIALIZATION,
   javax.safetycritical.annotate.Phase.RUN,
   javax.safetycritical.annotate.Phase.CLEANUP })
public java.lang.StackTraceElement[] getStackTrace( )
```

Get the stack trace created by fillInStackTrace for this Throwable as an array of StackTraceElements.

The stack trace does not need to contain entries for all methods that are actually on the call stack, the infrastructure may decide to skip some stack trace entries. Even an empty array is a valid result of this function.

Repeated calls of this function without intervening calls to fillInStackTrace will return the same result.

When memory areas are used, and this Throwable was allocated in a different memory area than the current allocation context, the resulting stack trace will be allocated in either the same memory area this was allocated in or the current memory area, depending on which is the least deeply nested, thereby creating objects that are assignment compatible with both areas.

returns array representing the stack trace, it is never null.

```
@SCJMayAllocate({})
@SCJAllowed
@SCJMaySelfSuspend(false)
@SCJPhase({
   javax.safetycritical.annotate.Phase.STARTUP,
   javax.safetycritical.annotate.Phase.INITIALIZATION,
   javax.safetycritical.annotate.Phase.RUN,
   javax.safetycritical.annotate.Phase.CLEANUP })
public java.lang.Throwable initCause(Throwable causingThrowable)
```

Initializes the cause to the given Throwable in the schedulable object's local memory.

causingThrowable — the reason why this Throwable gets Thrown.

returns the reference to this Throwable.

Throws IllegalArgumentException when the cause is this Throwable itself.

```
@SCJMayAllocate({})
@SCJAllowed
@SCJMaySelfSuspend(false)
@SCJPhase({
   javax.safetycritical.annotate.Phase.STARTUP,
   javax.safetycritical.annotate.Phase.INITIALIZATION,
   javax.safetycritical.annotate.Phase.RUN,
   javax.safetycritical.annotate.Phase.CLEANUP })
public java.lang.Throwable initMessage(String message)
```

Set the message in the schedulable object's local storage. This is the only method that is not also defined in Throwable.

message — is the text to set.

```
@SCJMayAllocate({})
@SCJAllowed
@SCJMaySelfSuspend(false)
@SCJPhase({
    javax.safetycritical.annotate.Phase.STARTUP,
    javax.safetycritical.annotate.Phase.INITIALIZATION,
    javax.safetycritical.annotate.Phase.RUN,
    javax.safetycritical.annotate.Phase.CLEANUP })
public void printStackTrace(PrintStream stream)
```

Print the stack trace of this Throwable to the given stream.

The printed stack trace contains the result of toString() as the first line followed by one line for each stack trace element that contains the name of the method or constructor, optionally followed by the source file name and source file line number when available.

stream — the stream to print to.

```
@SCJMayAllocate({})
@SCJAllowed
@SCJMaySelfSuspend(false)
@SCJPhase({
    javax.safetycritical.annotate.Phase.STARTUP,
    javax.safetycritical.annotate.Phase.INITIALIZATION,
    javax.safetycritical.annotate.Phase.RUN,
    javax.safetycritical.annotate.Phase.CLEANUP })
public void printStackTrace( )
```

Print stack trace of this Throwable to System.err.

The printed stack trace contains the result of toString() as the first line followed by one line for each stack trace element that contains the name of the method or constructor, optionally followed by the source file name and source file line number when available.

```
@SCJMayAllocate({})
@SCJAllowed
@SCJMaySelfSuspend(false)
@SCJPhase({
    javax.safetycritical.annotate.Phase.STARTUP,
    javax.safetycritical.annotate.Phase.INITIALIZATION,
    javax.safetycritical.annotate.Phase.RUN,
    javax.safetycritical.annotate.Phase.CLEANUP })
public void setStackTrace(StackTraceElement [] new_stackTrace)
  throws java.lang.NullPointerException
```

This method allows overriding the stack trace that was filled during construction of this object. It is intended to be used in a serialization context when the stack trace of a remote exception should be treated like a local.

new_stackTrace — the stack trace to replace be used.

Throws NullPointerException when new_stackTrace or any element of new_stackTrace is null.

D.2.8 INTERFACE **Timable**

@SCJAllowed
public interface Timable **extends** javax.realtime.Releasable

An interface for RealtimeThread to indicate that it can be associated with a clock and be suspended waiting for timing events based on that clock. This interface make use of some interfaces and classes in the RTSJ that are not visible to the SCJ. They are, therefore, not presented in this specification.

D.3 Classes

D.3.1 CLASS **AbsoluteTime**

@SCJAllowed
public class AbsoluteTime **extends** javax.realtime.HighResolutionTime

An object that represents a specific point in time given by milliseconds plus nanoseconds past some point in time fixed by its associated clock. For the default real-time clock the fixed point is the implementation dependent Epoch.

The correctness of the Epoch as a time base depends on the real-time clock synchronization with an external world time reference. This representation was designed to be compatible with the standard Java representation of an absolute time in the java.util.Date class.

Caution: This class is explicitly unsafe in multithreaded situations when it is being changed. No synchronization is done. It is assumed that users of this class who are mutating instances will be doing their own synchronization at a higher level.

Constructors

```
@SCJAllowed
@SCJPhase({
   javax.safetycritical.annotate.Phase.INITIALIZATION,
   javax.safetycritical.annotate.Phase.RUN,
   javax.safetycritical.annotate.Phase.CLEANUP})
@SCJMaySelfSuspend(false)
@SCJMayAllocate({})
public AbsoluteTime(long millis, int nanos, Chronograph clock)
```

Construct an AbsoluteTime object with time millisecond and nanosecond components past the Epoch for clock.

The value of the AbsoluteTime instance is based on the parameter millis plus the parameter nanos. The construction is subject to millis and nanos parameters normalization. If there is an overflow in the millisecond component when normalizing then an IllegalArgumentException will be thrown. If after normalization the time object is negative then the time represented by this is time before the Epoch.

The clock association is made with the clock parameter.

This constructor requires that the ”clock” parameter resides in a scope that encloses the scope of the ”this” argument.

millis — The desired value for the millisecond component of this. The actual value is the result of parameter normalization.

nanos — The desired value for the nanosecond component of this. The actual value is the result of parameter normalization.

clock — The chronograph providing the association for the newly constructed object. If clock is null the association is made with the real-time clock.

Throws IllegalArgumentException if there is an overflow in the millisecond component when normalizing.

```
@SCJAllowed
@SCJPhase({
   javax.safetycritical.annotate.Phase.INITIALIZATION,
   javax.safetycritical.annotate.Phase.RUN,
   javax.safetycritical.annotate.Phase.CLEANUP})
@SCJMaySelfSuspend(false)
@SCJMayAllocate({})
public AbsoluteTime(long millis, int nanos)
```

This constructor behaves the same as calling AbsoluteTime(millis, nanos, null).

```
@SCJAllowed
@SCJPhase({
   javax.safetycritical.annotate.Phase.INITIALIZATION,
   javax.safetycritical.annotate.Phase.RUN,
   javax.safetycritical.annotate.Phase.CLEANUP})
@SCJMaySelfSuspend(false)
@SCJMayAllocate({})
public AbsoluteTime( )
```

This constructor behaves the same as calling AbsoluteTime(0, 0, null).

```
@SCJAllowed
@SCJPhase({
   javax.safetycritical.annotate.Phase.INITIALIZATION,
   javax.safetycritical.annotate.Phase.RUN,
   javax.safetycritical.annotate.Phase.CLEANUP})
@SCJMaySelfSuspend(false)
@SCJMayAllocate({})
public AbsoluteTime(Chronograph clock)
```

This constructor behaves the same as calling AbsoluteTime(0, 0, clock).

```
@SCJAllowed
@SCJPhase({
   javax.safetycritical.annotate.Phase.INITIALIZATION,
   javax.safetycritical.annotate.Phase.RUN,
   javax.safetycritical.annotate.Phase.CLEANUP})
@SCJMaySelfSuspend(false)
@SCJMayAllocate({})
public AbsoluteTime(AbsoluteTime time)
```

Make a new AbsoluteTime object from the given AbsoluteTime object. The new object will have the same clock association as the time parameter.

This constructor requires that the ”time” parameter resides in a scope that encloses the scope of the ”this” argument.

time — The AbsoluteTime object which is the source for the copy.

Throws IllegalArgumentException if the time parameter is null.

Methods

```
@SCJAllowed
@SCJPhase({
   javax.safetycritical.annotate.Phase.INITIALIZATION,
```

```
    javax.safetycritical.annotate.Phase.RUN,
    javax.safetycritical.annotate.Phase.CLEANUP})
@SCJMaySelfSuspend(false)
@SCJMayAllocate({javax.safetycritical.annotate.AllocationContext.CURRENT})
public javax.realtime.AbsoluteTime add(long millis, int nanos, AbsoluteTime dest)
```

Return an object containing the value resulting from adding millis and nanos to the values from this and normalizing the result. If dest is not null, the result is placed there and returned. Otherwise, a new object is allocated for the result.

The result will have the same clock association as this, and the clock association of dest is ignored.

An ArithmeticException is thrown if the result does not fit in the normalized format.

millis — The number of milliseconds to be added to this.

nanos — The number of nanoseconds to be added to this.

dest — If dest is not null, the result is placed there. Otherwise, a new object is allocated for the result.

returns the value of dest if dest is not null, otherwise a new object representing the result.

Throws ArithmeticException if the result does not fit in the normalized format.

```
@SCJAllowed
@SCJPhase({
    javax.safetycritical.annotate.Phase.INITIALIZATION,
    javax.safetycritical.annotate.Phase.RUN,
    javax.safetycritical.annotate.Phase.CLEANUP})
@SCJMaySelfSuspend(false)
@SCJMayAllocate({javax.safetycritical.annotate.AllocationContext.CURRENT})
public javax.realtime.AbsoluteTime add(RelativeTime time, AbsoluteTime dest)
```

Return an object containing the value resulting from adding time to the value of this and normalizing the result. If dest is not null, the result is placed there and returned. Otherwise, a new object is allocated for the result.

The clock associated with this and the clock associated with the time parameter must be the same, and such association is used for the result.

The clock associated with the dest parameter is ignored.

An IllegalArgumentException is thrown if the clock associated with this and the clock associated with the time parameter are different.

An IllegalArgumentException is thrown if the time parameter is null.

An ArithmeticException is thrown if the result does not fit in the normalized format.

time — The time to add to this.

dest — If dest is not null, the result is placed there and returned. Otherwise, a new object is allocated for the result.

returns the value of dest if dest is not null, otherwise a new object representing the result.

Throws IllegalArgumentException if the clock associated with this and the clock associated with the time parameter are different, or when the time parameter is null.

Throws ArithmeticException if the result does not fit in the normalized format.

```
@SCJAllowed
@SCJPhase({
    javax.safetycritical.annotate.Phase.INITIALIZATION,
    javax.safetycritical.annotate.Phase.RUN,
    javax.safetycritical.annotate.Phase.CLEANUP})
@SCJMaySelfSuspend(false)
@SCJMayAllocate({javax.safetycritical.annotate.AllocationContext.CURRENT})
public javax.realtime.AbsoluteTime add(RelativeTime time)
```

Create a new instance of AbsoluteTime representing the result of adding time to the value of this and normalizing the result.

The clock associated with this and the clock associated with the time parameter must be the same, and such association is used for the result.

An IllegalArgumentException is thrown if the clock associated with this and the clock associated with the time parameter are different.

An IllegalArgumentException is thrown if the time parameter is null.

An ArithmeticException is thrown if the result does not fit in the normalized format.

time — The time to add to this.

returns A new AbsoluteTime object whose time is the normalization of this plus the parameter time.

Throws IllegalArgumentException if the clock associated with this and the clock associated with the time parameter are different, or when the time parameter is null.

Throws ArithmeticException if the result does not fit in the normalized format.

```
@SCJAllowed
@SCJPhase({
    javax.safetycritical.annotate.Phase.INITIALIZATION,
    javax.safetycritical.annotate.Phase.RUN,
    javax.safetycritical.annotate.Phase.CLEANUP})
```

```
@SCJMaySelfSuspend(false)
@SCJMayAllocate({javax.safetycritical.annotate.AllocationContext.CURRENT})
public javax.realtime.AbsoluteTime add(long millis, int nanos)
```

Create a new object representing the result of adding millis and nanos to the values from this and normalizing the result. The result will have the same clock association as this. An ArithmeticException is thrown if the result does not fit in the normalized format.

millis — The number of milliseconds to be added to this.

nanos — The number of nanoseconds to be added to this.

returns A new AbsoluteTime object whose time is the normalization of this plus millis and nanos.

Throws ArithmeticException if the result does not fit in the normalized format.

```
@SCJAllowed
@SCJMaySelfSuspend(false)
@SCJMayAllocate({})
@Override
@SCJPhase({
    javax.safetycritical.annotate.Phase.STARTUP,
    javax.safetycritical.annotate.Phase.INITIALIZATION,
    javax.safetycritical.annotate.Phase.RUN,
    javax.safetycritical.annotate.Phase.CLEANUP })
public int compareTo(AbsoluteTime time)
```

Compares this object with the specified object for order.

returns a negative integer, zero, or a positive integer as this object is less than, equal to, or greater than the specified object.

Throws ClassCastException if the time parameter is not of the same class as this.

Throws IllegalArgumentException if the time parameter is not associated with the same clock as this, or when the time parameter is null.

```
@SCJAllowed
@SCJPhase({
    javax.safetycritical.annotate.Phase.INITIALIZATION,
    javax.safetycritical.annotate.Phase.RUN,
    javax.safetycritical.annotate.Phase.CLEANUP})
@SCJMaySelfSuspend(false)
@SCJMayAllocate({javax.safetycritical.annotate.AllocationContext.CURRENT})
public javax.realtime.RelativeTime subtract(AbsoluteTime time, RelativeTime dest)
```

Return an object containing the value resulting from subtracting time from the value of this and normalizing the result. If dest is not null, the result is placed there and returned. Otherwise, a new object is allocated for the result. The clock associated with this and the clock associated with the time parameter must be the same, and such association is used for the result.

The clock associated with the dest parameter is ignored.

An IllegalArgumentException is thrown if the clock associated with this and the clock associated with the time parameter are different. An IllegalArgument-Exception is thrown if the time parameter is null.

An ArithmeticException is thrown if the result does not fit in the normalized format.

time — The time to subtract from this.

dest — If dest is not null, the result is placed there and returned. Otherwise, a new object is allocated for the result.

returns the value of dest if dest is not null, otherwise a new object representing the result.

Throws IllegalArgumentException if the clock associated with this and the clock associated with the time parameter are different, or when the time parameter is null.

Throws ArithmeticException if the result does not fit in the normalized format.

```
@SCJAllowed
@SCJPhase({
    javax.safetycritical.annotate.Phase.INITIALIZATION,
    javax.safetycritical.annotate.Phase.RUN,
    javax.safetycritical.annotate.Phase.CLEANUP})
@SCJMaySelfSuspend(false)
@SCJMayAllocate({javax.safetycritical.annotate.AllocationContext.CURRENT})
public javax.realtime.AbsoluteTime subtract(RelativeTime time, AbsoluteTime dest)
```

Return an object containing the value resulting from subtracting time from the value of this and normalizing the result.

time — The time to subtract from this.

dest — If dest is not null, the result is placed there and returned. Otherwise, a new object is allocated for the result.

returns the value of dest if dest is not null, otherwise a new object representing the result.

```
@SCJAllowed
@SCJPhase({
   javax.safetycritical.annotate.Phase.INITIALIZATION,
   javax.safetycritical.annotate.Phase.RUN,
   javax.safetycritical.annotate.Phase.CLEANUP})
@SCJMaySelfSuspend(false)
@SCJMayAllocate({javax.safetycritical.annotate.AllocationContext.CURRENT})
public javax.realtime.AbsoluteTime subtract(RelativeTime time)
```

Create a new instance of AbsoluteTime representing the result of subtracting time from the value of this and normalizing the result.

The clock associated with this and the clock associated with the time parameter must be the same, and such association is used for the result.

An IllegalArgumentException is thrown if the clock associated with this and the clock associated with the time parameter are different. An IllegalArgument-Exception is thrown if the time parameter is null.

An ArithmeticException is thrown if the result does not fit in the normalized format.

time — The time to subtract from this.

returns A new AbsoluteTime object whose time is the normalization of this minus the parameter time.

Throws IllegalArgumentException if the clock associated with this and the clock associated with the time parameter are different, or when the time parameter is null.

Throws ArithmeticException if the result does not fit in the normalized format.

```
@SCJAllowed
@SCJPhase({
   javax.safetycritical.annotate.Phase.INITIALIZATION,
   javax.safetycritical.annotate.Phase.RUN,
   javax.safetycritical.annotate.Phase.CLEANUP})
@SCJMaySelfSuspend(false)
@SCJMayAllocate({javax.safetycritical.annotate.AllocationContext.CURRENT})
public javax.realtime.RelativeTime subtract(AbsoluteTime time)
```

Create a new instance of RelativeTime representing the result of subtracting time from the value of this and normalizing the result.

The clock associated with this and the clock associated with the time parameter must be the same, and such association is used for the result.

An IllegalArgumentException is thrown if the clock associated with this and the clock associated with the time parameter are different.

An IllegalArgumentException is thrown if the time parameter is null.

An ArithmeticException is thrown if the result does not fit in the normalized format.

time — The time to subtract from this.

returns A new RelativeTime object whose time is the normalization of this minus the AbsoluteTime parameter time.

Throws IllegalArgumentException if the clock associated with this and the clock associated with the time parameter are different, or when the time parameter is null.

Throws ArithmeticException if the result does not fit in the normalized format.

D.3.2 CLASS **Affinity**

@SCJAllowed
public final class Affinity **extends** java.lang.Object

This class is the API for all processor-affinity-related aspects of the RTSJ that are relevant to the SCJ. It includes a factory that generates Affinity objects. The explicit setting of the affinity of SCJ managed schedulables is performed during its mission initialisation phase when the managed schedulable is registered. If no affinity is set, the managed schedulable inherits the affinity of its mission sequencer.

An affinity set is a set of processors that can be associated with a real-time thread or async event handler. For SCJ, an affinity set is associated to a managed schedulable. Each implementation supports an array of predefined affinity sets. They can be used either to reflect the scheduling arrangement of the underlying OS or they can be used by the system designer to impose defaults for, schedulable objects. An application is only allowed to dynamically create new affinity sets with cardinality of one. This restriction reflects the concern that not all operating systems will support multiprocessor affinity sets.

The processor membership of an affinity set is immutable. The schedulable object associations to an affinity set are mutable.

The internal representation of an affinity set in an Affinity instance is not specified, but the representation that is used to communicate with this class is a BitSet where each bit corresponds to a logical processor ID. The relationship between logical and physical processors is implementation defined, and may differ from one implementation to another.

The affinity set factory may be used to create affinity sets with a single processor member at any time, though this operation only supports processor members that are valid as the processor affinity for a schedulable object (at the time

of the affinity set's creation.) The factory cannot create an affinity set with more than one processor member, but such affinity sets are supported. They may be internally created by the SCJ infrastructure, probably at start up time.

The set of affinity sets created by the infrastructure at start up (the predefined set) is visible through the getPredefinedAffinities(Affinity[]) method. In SCJ the initial mission sequencer has an affinity equal to getPredefinedAffinities(Affinity[])[0]; that is the first element of the returned array

External changes to the set of processors available to the SCJ infrastructure is likely to cause serious trouble ranging from violation of assumptions underlying schedulability analysis to freezing the entire SCJ program, so if a system is capable of such manipulation it should not exercise it on SCJ processes.

There is no public constructor for this class. All instances must be generated by the factory method (generate).

Methods

```
@SCJAllowed(javax.safetycritical.annotate.Level.LEVEL_2)
@SCJMaySelfSuspend(false)
@SCJMayAllocate({javax.safetycritical.annotate.AllocationContext.CURRENT})
@SCJPhase({
   javax.safetycritical.annotate.Phase.STARTUP,
   javax.safetycritical.annotate.Phase.INITIALIZATION,
   javax.safetycritical.annotate.Phase.RUN,
   javax.safetycritical.annotate.Phase.CLEANUP })
public static final javax.realtime.Affinity generate(BitSet bitSet)
```

Returns an Affinity set with the affinity bitSet and no associations.

Platforms that support specific affinity sets will register those Affinity instances with Affinity. They appear in the arrays returned bygetPredefinedAffinities() and getPredefinedAffinities(Affinity[]).

bitSet — The set of processors associated with the generated Affinity.

returns The resulting Affinity.

Throws NullPointerException when bitSet is null.

Throws IllegalArgumentException when bitSet does not refer to a valid set of processors, where "valid" is defined as the bitset from a pre-defined affinity set, or a bitset of cardinality one containing a processor from the set returned by getAvailableProcessors(). The definition of "valid set of processors" is system dependent; however, every set consisting of one valid processor makes up a valid bit set, and every bit set corresponding to a pre-defined affinity set is valid.

```
@SCJAllowed(javax.safetycritical.annotate.Level.LEVEL_1)
@SCJMaySelfSuspend(false)
@SCJMayAllocate({javax.safetycritical.annotate.AllocationContext.CURRENT})
@SCJPhase({
   javax.safetycritical.annotate.Phase.STARTUP,
   javax.safetycritical.annotate.Phase.INITIALIZATION,
   javax.safetycritical.annotate.Phase.RUN,
   javax.safetycritical.annotate.Phase.CLEANUP })
public static final java.util.BitSet getAvailableProcessors(BitSet dest)
```

In systems where the set of processors available to a process is dynamic (e.g., because of system management operations or because of fault tolerance capabilities), the result of this operation shall reflect the processsors that are currently allocated to the SCJ infrastructure and are currently available to execute tasks.

dest — If dest is non-null, use dest for the returned value. If it is null, create a new BitSet.

returns the set of processors representing the set of processors currently valid for the bitset argument to generate(BitSet).

```
@SCJAllowed(javax.safetycritical.annotate.Level.LEVEL_1)
@SCJMaySelfSuspend(false)
@SCJMayAllocate({javax.safetycritical.annotate.AllocationContext.CURRENT})
@SCJPhase({
   javax.safetycritical.annotate.Phase.STARTUP,
   javax.safetycritical.annotate.Phase.INITIALIZATION,
   javax.safetycritical.annotate.Phase.RUN,
   javax.safetycritical.annotate.Phase.CLEANUP })
public static final java.util.BitSet getAvailableProcessors( )
```

This method is equivalent to getAvailableProcessors(null).

returns the set of processors available to the application.

```
@SCJAllowed(javax.safetycritical.annotate.Level.LEVEL_1)
@SCJMaySelfSuspend(false)
@SCJMayAllocate({javax.safetycritical.annotate.AllocationContext.CURRENT})
@SCJPhase({
   javax.safetycritical.annotate.Phase.STARTUP,
   javax.safetycritical.annotate.Phase.INITIALIZATION,
   javax.safetycritical.annotate.Phase.RUN,
   javax.safetycritical.annotate.Phase.CLEANUP })
public static final javax.realtime.Affinity[] getPredefinedAffinities( )
```

Equivalent to invoking getPredefinedAffinitySets(null).

returns an array of the pre-defined affinity sets.

```
@SCJAllowed(javax.safetycritical.annotate.Level.LEVEL_1)
@SCJMaySelfSuspend(false)
@SCJMayAllocate({javax.safetycritical.annotate.AllocationContext.CURRENT})
@SCJPhase({
    javax.safetycritical.annotate.Phase.STARTUP,
    javax.safetycritical.annotate.Phase.INITIALIZATION,
    javax.safetycritical.annotate.Phase.RUN,
    javax.safetycritical.annotate.Phase.CLEANUP })
public static final javax.realtime.Affinity[] getPredefinedAffinities(
  Affinity [] dest)
```

Return an array containing all affinity sets that were predefined by the infrastructure.

dest — The destination array, or null.

returns dest or a newly created array if dest was null, populated with references to the pre-defined affinity sets.

If dest has excess entries, they are filled with null.

Throws IllegalArgumentException when dest is not large enough.

```
@SCJAllowed(javax.safetycritical.annotate.Level.LEVEL_1)
@SCJMaySelfSuspend(false)
@SCJMayAllocate({})
@SCJPhase({
    javax.safetycritical.annotate.Phase.STARTUP,
    javax.safetycritical.annotate.Phase.INITIALIZATION,
    javax.safetycritical.annotate.Phase.RUN,
    javax.safetycritical.annotate.Phase.CLEANUP })
public static final int getPredefinedAffinitiesCount( )
```

Return the minimum array size required to store references to all the predefined processor affinity sets.

returns the minimum array size required to store references to all the predefined affinity sets.

```
@SCJAllowed(javax.safetycritical.annotate.Level.LEVEL_1)
@SCJMaySelfSuspend(false)
@SCJMayAllocate({javax.safetycritical.annotate.AllocationContext.CURRENT})
@SCJPhase({
    javax.safetycritical.annotate.Phase.STARTUP,
    javax.safetycritical.annotate.Phase.INITIALIZATION,
    javax.safetycritical.annotate.Phase.RUN,
    javax.safetycritical.annotate.Phase.CLEANUP })
public final java.util.BitSet getProcessors(BitSet dest)
```

Return a BitSet representing the processor affinity set of this Affinity.

dest — Set dest to the BitSet value. If dest is null, create a new BitSet in the current allocation context.

returns a BitSet representing the processor affinity set of this Affinity.

```
@SCJAllowed(javax.safetycritical.annotate.Level.LEVEL_1)
@SCJMaySelfSuspend(false)
@SCJMayAllocate({javax.safetycritical.annotate.AllocationContext.CURRENT})
@SCJPhase({
    javax.safetycritical.annotate.Phase.STARTUP,
    javax.safetycritical.annotate.Phase.INITIALIZATION,
    javax.safetycritical.annotate.Phase.RUN,
    javax.safetycritical.annotate.Phase.CLEANUP })
public final java.util.BitSet getProcessors( )
```

Return a BitSet representing the processor affinity set for this Affinity.

returns a newly created BitSet representing this Affinity.

```
@SCJAllowed(javax.safetycritical.annotate.Level.LEVEL_1)
@SCJMaySelfSuspend(false)
@SCJMayAllocate({})
@SCJPhase({
    javax.safetycritical.annotate.Phase.STARTUP,
    javax.safetycritical.annotate.Phase.INITIALIZATION,
    javax.safetycritical.annotate.Phase.RUN,
    javax.safetycritical.annotate.Phase.CLEANUP })
public final boolean isProcessorInSet(int processorNumber)
```

Ask whether a processor is included in this affinity set.

processorNumber — is a logical processor number

returns true if and only if processorNumber is a member of this affinity set.

D.3.3 CLASS AperiodicParameters

```
@SCJAllowed(javax.safetycritical.annotate.Level.LEVEL_1)
public class AperiodicParameters extends javax.realtime.ReleaseParameters
```

SCJ supports no detection of minimum inter-arrival time violations, therefore only aperiodic parameters are needed. Hence the RTSJ SporadicParameters class is absent. Deadline miss detection is supported.

The RTSJ supports a queue for storing the arrival of release events is order to enable bursts of events to be handled. This queue is of length 1 in SCJ. The RTSJ also enables different responses to the queue overflowing. In SCJ the overflow behavior is to overwrite the pending release event if there is one.

Constructors

```
@SCJAllowed(javax.safetycritical.annotate.Level.LEVEL_1)
@SCJMaySelfSuspend(false)
@SCJMayAllocate({})
@SCJPhase({
    javax.safetycritical.annotate.Phase.STARTUP,
    javax.safetycritical.annotate.Phase.INITIALIZATION,
    javax.safetycritical.annotate.Phase.RUN,
    javax.safetycritical.annotate.Phase.CLEANUP })
public AperiodicParameters(RelativeTime deadline, AsyncEventHandler missHandler)
```

Construct a new AperiodicParameters object within the current memory area.

deadline — is an offset from the release time by which the release should finish. A null deadline indicates that there is no deadline.

missHandler — is the AsynchronousEventHandler to be released if the associated schedulable object misses its deadline. A null parameter indicates that no handler should be released.

```
@SCJAllowed(javax.safetycritical.annotate.Level.LEVEL_1)
@SCJMaySelfSuspend(false)
@SCJMayAllocate({})
@SCJPhase({
    javax.safetycritical.annotate.Phase.STARTUP,
    javax.safetycritical.annotate.Phase.INITIALIZATION,
    javax.safetycritical.annotate.Phase.RUN,
    javax.safetycritical.annotate.Phase.CLEANUP })
public AperiodicParameters( )
```

This constructor behaves the same as calling AperiodicParameters(null, null).

Methods

```
@SCJAllowed
@Override
@SCJMaySelfSuspend(false)
@SCJMayAllocate({javax.safetycritical.annotate.AllocationContext.CURRENT})
@SCJPhase({
    javax.safetycritical.annotate.Phase.STARTUP,
    javax.safetycritical.annotate.Phase.INITIALIZATION,
    javax.safetycritical.annotate.Phase.RUN,
    javax.safetycritical.annotate.Phase.CLEANUP })
public java.lang.Object clone( )
```

Create a clone of this AperiodicParameters object.

D.3.4 CLASS AsyncBaseEventHandler

@SCJAllowed
public abstract class AsyncBaseEventHandler **implements**
javax.realtime.Schedulable **extends** java.lang.Object

This is the base class for all asynchronous event handlers. In SCJ, this is an empty class.

Methods

@SCJAllowed
@SCJPhase({javax.safetycritical.annotate.Phase.RUN})
@SCJMayAllocate({})
@SCJMaySelfSuspend(false)
public void interrupt( )

Behaves as if {@code Thread.interrupt()} were called on the implementation thread underlying this Schedulable. throws IllegalSchedulableStateException when {@code this} is not currently releasable, i.e., its start method has not been called, or it has terminated.

@SCJAllowed
@SCJPhase({javax.safetycritical.annotate.Phase.RUN})
@SCJMayAllocate({})
@SCJMaySelfSuspend(false)
public boolean isInterrupted( )

Determines whether or not any interrupt is pending.

returns {@code true} when and only when the interrupt is pending.

D.3.5 CLASS AsyncEventHandler

@SCJAllowed
public class AsyncEventHandler **extends** javax.realtime.AsyncBaseEventHandler

In SCJ, all asynchronous events have their handlers bound to a thread when they are created (during the initialization phase). The binding is permanent. Thus, the AsyncEventHandler constructors are hidden from public view in the SCJ specification.

Methods

```
@SCJAllowed(javax.safetycritical.annotate.Level.SUPPORT)
@SCJPhase({javax.safetycritical.annotate.Phase.RUN})
@SCJMayAllocate({
   javax.safetycritical.annotate.AllocationContext.CURRENT,
   javax.safetycritical.annotate.AllocationContext.INNER,
   javax.safetycritical.annotate.AllocationContext.OUTER})
@SCJMaySelfSuspend(true)
public void handleAsyncEvent( )
```

This method must be overridden by the application to provide the handling code. Note that this method shall not self-suspend when called in a Level 0 mission.

```
@SCJAllowed(javax.safetycritical.annotate.Level.SUPPORT)
@SCJPhase({javax.safetycritical.annotate.Phase.RUN})
@Override
@SCJMayAllocate({
   javax.safetycritical.annotate.AllocationContext.CURRENT,
   javax.safetycritical.annotate.AllocationContext.INNER,
   javax.safetycritical.annotate.AllocationContext.OUTER})
@SCJMaySelfSuspend(true)
public final void run( )
```

This method is used by the SCJ infrastructure. It should not be called by the application.

D.3.6 CLASS AsyncLongEventHandler

```
@SCJAllowed
public class AsyncLongEventHandler extends
    javax.realtime.AsyncBaseEventHandler
```

In SCJ, all asynchronous events must have their handlers bound when they are created (during the initialization phase). The binding is permanent. Thus, the AsyncLongEventHandler constructors are hidden from public view in the SCJ specification. This class differs from AsyncEventHandler in that when it is fired, a long integer is provided for use by the released event handler(s).

Methods

```
@SCJAllowed(javax.safetycritical.annotate.Level.SUPPORT)
@SCJPhase({javax.safetycritical.annotate.Phase.RUN})
@SCJMayAllocate({
   javax.safetycritical.annotate.AllocationContext.CURRENT,
```

```
    javax.safetycritical.annotate.AllocationContext.INNER,
    javax.safetycritical.annotate.AllocationContext.OUTER})
@SCJMaySelfSuspend(true)
public void handleAsyncEvent(long data)
```

This method must be overridden by the application to provide the handling code. Note that this method shall not self-suspend when called in a Level 0 mission.

```
@SCJAllowed(javax.safetycritical.annotate.Level.SUPPORT)
@SCJPhase({javax.safetycritical.annotate.Phase.RUN})
@Override
@SCJMayAllocate({
    javax.safetycritical.annotate.AllocationContext.CURRENT,
    javax.safetycritical.annotate.AllocationContext.INNER,
    javax.safetycritical.annotate.AllocationContext.OUTER})
@SCJMaySelfSuspend(true)
public final void run( )
```

This method is used by the SCJ infrastructure. It should not be called by the application.

D.3.7 CLASS BoundAsyncEventHandler

```
@SCJAllowed
public class BoundAsyncEventHandler implements
      javax.realtime.BoundAsyncBaseEventHandler extends
      javax.realtime.AsyncEventHandler
```

The BoundAsyncEventHandler class is a base class inherited from RTSJ. None of its methods or constructors are publicly available.

Methods

```
@SCJAllowed(javax.safetycritical.annotate.Level.LEVEL_1)
@SCJMaySelfSuspend(false)
@SCJPhase({
    javax.safetycritical.annotate.Phase.STARTUP,
    javax.safetycritical.annotate.Phase.INITIALIZATION,
    javax.safetycritical.annotate.Phase.RUN,
    javax.safetycritical.annotate.Phase.CLEANUP})
@SCJMayAllocate({})
@Override
public javax.realtime.Affinity getAffinity( )
```

```
@SCJAllowed(javax.safetycritical.annotate.Level.LEVEL_1)
@SCJMaySelfSuspend(false)
@SCJPhase({
   javax.safetycritical.annotate.Phase.STARTUP,
   javax.safetycritical.annotate.Phase.INITIALIZATION})
@SCJMayAllocate({})
@Override
public void setAffinity(Affinity set)
  throws java.lang.IllegalArgumentException,
     javax.realtime.ProcessorAffinityException, java.lang.NullPointerException
```

D.3.8 CLASS BoundAsyncLongEventHandler

```
@SCJAllowed
public class BoundAsyncLongEventHandler implements
     javax.realtime.BoundAsyncBaseEventHandler extends
     javax.realtime.AsyncLongEventHandler
```

The BoundAsyncLongEventHandler is a base class inherited from RTSJ. None of its methods or constructors are publicly available. This class differs from BoundAsyncEventHandler in that when it is released, a long integer is provided for use by the released event handler(s).

Methods

```
@SCJAllowed(javax.safetycritical.annotate.Level.LEVEL_1)
@SCJMaySelfSuspend(false)
@SCJPhase({
   javax.safetycritical.annotate.Phase.STARTUP,
   javax.safetycritical.annotate.Phase.INITIALIZATION})
@SCJMayAllocate({})
@Override
public javax.realtime.Affinity getAffinity( )
```

Note: since this is only used by infrastructure, we don't specify the MemoryAreaEncloses relationships. public BoundAsyncLongEventHandler(SchedulingParameters scheduling, ReleaseParameters release, MemoryParameters memory, MemoryArea area, ProcessingGroupParameters group, boolean noheap, Runnable logic) { super(scheduling, release, memory, area, group, noheap, logic); }

```
@SCJAllowed(javax.safetycritical.annotate.Level.LEVEL_1)
@SCJMaySelfSuspend(false)
@SCJPhase({
   javax.safetycritical.annotate.Phase.STARTUP,
   javax.safetycritical.annotate.Phase.INITIALIZATION,
```

```
    javax.safetycritical.annotate.Phase.RUN,
    javax.safetycritical.annotate.Phase.CLEANUP})
@SCJMayAllocate({})
@Override
public void setAffinity(Affinity set)
  throws java.lang.IllegalArgumentException,
      javax.realtime.ProcessorAffinityException, java.lang.NullPointerException
```

D.3.9 CLASS Clock

```
@SCJAllowed
public abstract class Clock implements javax.realtime.Chronograph extends
      java.lang.Object
```

A clock is a chronograph that also manages time events (also called alarms) that can be queued on it and that will cause an event handler to be released when their appointed time is reached.

The Clock instance returned by getRealtimeClock may be used in any context that requires a clock.

HighResolutionTime instances that use application-defined clocks are valid for all APIs in SCJ that take HighResolutionTime time types as parameters.

Constructors

```
@SCJMayAllocate({})
@SCJAllowed(javax.safetycritical.annotate.Level.LEVEL_1)
@SCJMaySelfSuspend(false)
@SCJPhase({
    javax.safetycritical.annotate.Phase.STARTUP,
    javax.safetycritical.annotate.Phase.INITIALIZATION,
    javax.safetycritical.annotate.Phase.RUN,
    javax.safetycritical.annotate.Phase.CLEANUP })
public Clock( )
```

Constructor for the abstract class.

Methods

```
@SCJAllowed(javax.safetycritical.annotate.Level.LEVEL_1)
@SCJMayAllocate({})
@SCJMaySelfSuspend(false)
@SCJPhase({
    javax.safetycritical.annotate.Phase.STARTUP,
    javax.safetycritical.annotate.Phase.INITIALIZATION,
```

```
    javax.safetycritical.annotate.Phase.RUN,
    javax.safetycritical.annotate.Phase.CLEANUP })
protected abstract void clearAlarm( )
```

Implemented by subclasses to cancel the current outstanding alarm.

```
@SCJMayAllocate({javax.safetycritical.annotate.AllocationContext.CURRENT})
@SCJAllowed
@SCJMaySelfSuspend(false)
@SCJPhase({
    javax.safetycritical.annotate.Phase.STARTUP,
    javax.safetycritical.annotate.Phase.INITIALIZATION,
    javax.safetycritical.annotate.Phase.RUN,
    javax.safetycritical.annotate.Phase.CLEANUP })
public abstract javax.realtime.RelativeTime getDrivePrecision( )
```

Gets the precision of the clock for driving events, It is the same as calling getDrivePrecision(null) .

returns a value representing the drive precision.

```
@SCJMayAllocate({javax.safetycritical.annotate.AllocationContext.CURRENT})
@SCJAllowed
@SCJMaySelfSuspend(false)
@SCJPhase({
    javax.safetycritical.annotate.Phase.STARTUP,
    javax.safetycritical.annotate.Phase.INITIALIZATION,
    javax.safetycritical.annotate.Phase.RUN,
    javax.safetycritical.annotate.Phase.CLEANUP })
public abstract javax.realtime.RelativeTime getDrivePrecision(RelativeTime dest)
```

Gets the precision of the clock for driving events, defined as the nominal interval between ticks that can trigger an event. This is the resolution that shall be used for all scheduling decisions based on this clock. The result may be larger than that of getQueryPrecision() .

dest — returns the precision in dest. When dest is null, it allocates a new RelativeTime instance to hold the returned value.

returns the value of dest if dest is not null, otherwise a new object representing the drive precision.

```
@SCJMayAllocate({javax.safetycritical.annotate.AllocationContext.CURRENT})
@SCJAllowed(javax.safetycritical.annotate.Level.LEVEL_1)
@SCJMaySelfSuspend(false)
@SCJPhase({
```

```
    javax.safetycritical.annotate.Phase.STARTUP,
    javax.safetycritical.annotate.Phase.INITIALIZATION,
    javax.safetycritical.annotate.Phase.RUN,
    javax.safetycritical.annotate.Phase.CLEANUP })
public final javax.realtime.RelativeTime getEpochOffset( )
```

Determines the time on the real-time clock when this clcok was zero.

returns A newly allocated RelativeTime object in the current execution context with the real-time clock as its chronograph and containing the time when this chronograph was zero.

Throws UnsupportedOperationException when the clock does not have the concept of Epoch.

```
@SCJMayAllocate({javax.safetycritical.annotate.AllocationContext.CURRENT})
@SCJAllowed
@SCJMaySelfSuspend(false)
@SCJPhase({
    javax.safetycritical.annotate.Phase.STARTUP,
    javax.safetycritical.annotate.Phase.INITIALIZATION,
    javax.safetycritical.annotate.Phase.RUN,
    javax.safetycritical.annotate.Phase.CLEANUP })
public abstract javax.realtime.RelativeTime getQueryPrecision( )
```

Gets the precision of the time read, defined as the nominal interval between ticks. It is the same as calling getQueryPrecision(null) .

returns the value representing the read precision.

```
@SCJMayAllocate({javax.safetycritical.annotate.AllocationContext.CURRENT})
@SCJAllowed
@SCJMaySelfSuspend(false)
@SCJPhase({
    javax.safetycritical.annotate.Phase.STARTUP,
    javax.safetycritical.annotate.Phase.INITIALIZATION,
    javax.safetycritical.annotate.Phase.RUN,
    javax.safetycritical.annotate.Phase.CLEANUP })
public abstract javax.realtime.RelativeTime getQueryPrecision(RelativeTime dest)
```

Gets the precision of the time read, defined as the nominal interval between ticks. The result may be smaller than that of getDrivePrecision(), when the clock is tied to some system tick for releasing time events.

dest — returns the relative time value in dest. When dest is null, allocate a new RelativeTime instance to hold the returned value.

returns the value of dest if dest is not null, otherwise a new object representing the read precision.

```
@SCJAllowed
@SCJMaySelfSuspend(false)
@SCJMayAllocate({})
@SCJPhase({
   javax.safetycritical.annotate.Phase.STARTUP,
   javax.safetycritical.annotate.Phase.INITIALIZATION,
   javax.safetycritical.annotate.Phase.RUN,
   javax.safetycritical.annotate.Phase.CLEANUP })
public static javax.realtime.Clock getRealtimeClock( )
```

There is always at least one clock object available: the system real-time clock. This is the default Clock.

returns the singleton instance of the default Clock.

```
@SCJAllowed
@SCJMayAllocate({})
@SCJMaySelfSuspend(false)
@SCJPhase({
   javax.safetycritical.annotate.Phase.STARTUP,
   javax.safetycritical.annotate.Phase.INITIALIZATION,
   javax.safetycritical.annotate.Phase.RUN,
   javax.safetycritical.annotate.Phase.CLEANUP })
public abstract javax.realtime.AbsoluteTime getTime(AbsoluteTime dest)
```

Gets the current time in an existing object. The time represented by the given AbsoluteTime is changed at some time between the invocation of the method and the return of the method. This method will return an absolute time value that represents the clock's notion of the current absolute time. For clocks that do not measure calendar time, this absolute time may not represent a wall clock time.

dest — The instance of AbsoluteTime object that will be updated in place. The clock association of the dest parameter is overwritten. When dest is not null the returned object is associated with this clock. If dest is null, then nothing happens.

returns the instance of AbsoluteTime passed as a parameter, representing the current time, associated with this clock, or null if dest was null.

```
@SCJMayAllocate({javax.safetycritical.annotate.AllocationContext.CURRENT})
@SCJAllowed
@SCJMaySelfSuspend(false)
@SCJPhase({
   javax.safetycritical.annotate.Phase.STARTUP,
   javax.safetycritical.annotate.Phase.INITIALIZATION,
   javax.safetycritical.annotate.Phase.RUN,
   javax.safetycritical.annotate.Phase.CLEANUP })
public final javax.realtime.AbsoluteTime getTime( )
```

Gets the current time in a newly allocated object. This method will return an absolute time value that represents the clock's notion of an absolute time. For clocks that do not measure calendar time, this absolute time may not represent a wall clock time.

returns a newly allocated instance of AbsoluteTime in the current allocation context, representing the current time. The returned object is associated with this clock.

```
@SCJAllowed(javax.safetycritical.annotate.Level.LEVEL_1)
@SCJMayAllocate({})
@SCJMaySelfSuspend(false)
@SCJPhase({
   javax.safetycritical.annotate.Phase.STARTUP,
   javax.safetycritical.annotate.Phase.INITIALIZATION,
   javax.safetycritical.annotate.Phase.RUN,
   javax.safetycritical.annotate.Phase.CLEANUP })
protected abstract void setAlarm(long milliseconds, int nanoseconds)
```

Implemented by subclasses to set the time for the next alarm. If there is an alarm outstanding when called, it overwrites the old time. The milliseconds and nanoseconds are interpreted as an absolute time.

milliseconds — of the next alarm.

nanoseconds — of the next alarm.

```
@SCJAllowed(javax.safetycritical.annotate.Level.LEVEL_1)
@SCJMayAllocate({})
@SCJMaySelfSuspend(false)
@SCJPhase({
   javax.safetycritical.annotate.Phase.STARTUP,
   javax.safetycritical.annotate.Phase.INITIALIZATION,
   javax.safetycritical.annotate.Phase.RUN,
   javax.safetycritical.annotate.Phase.CLEANUP })
protected final void triggerAlarm( )
```

Called by a subclass to signal that the time of the next alarm has been reached.

D.3.10 CLASS ConfigurationParameters

```
@SCJAllowed
public class ConfigurationParameters extends java.lang.Object
```

Schedulable sizing parameters a way to specify various implementation-dependent parameters such as Java and native stack sizes, and to configure the statically allocated ThrowBoundaryError associated with a Schedulable.

Note that these parameters are immutable.

Constructors

```
@SCJAllowed
@SCJPhase({
    javax.safetycritical.annotate.Phase.INITIALIZATION,
    javax.safetycritical.annotate.Phase.RUN,
    javax.safetycritical.annotate.Phase.CLEANUP})
@SCJMaySelfSuspend(false)
@SCJMayAllocate({})
public ConfigurationParameters(int messageLength,
  int stackTraceLength,
  long [] sizes)
```

Creates a parameter object for initializing the state of a Schedulable. The parameters provide the data for this initialization.

messageLength — Memory space in bytes dedicated to the message associated with Schedulable objects created with these parameters' preallocated exceptions, plus references to the method names/identifiers in the stack trace. The value 0 indicates that no message should be stored. The value of -1 uses the system default.

stackTraceLength — Length of the stack trace buffer dedicated to Schedulable objects created with these parameters' preallocated exceptions, in frames. The amount of space this requires is implementation-specific. The value 0 indicates that no stack trace should be stored. The value of -1 uses the system default.

sizes — An array of implementation-specific values dictating memory parameters for Schedulable objects created with these parameters, such as maximum Java and native stack sizes. The sizes array will not be stored in the constructed object.

Throws IllegalArgumentException if messageLength or stackTraceLength is less than -1.

D.3.11 CLASS DeregistrationException

```
@SCJAllowed
public class DeregistrationException extends
      javax.realtime.StaticRuntimeException
```

The exception thrown when deregistering an InterruptServiceRoutine

Methods

```
@SCJMayAllocate({})
@SCJAllowed
@SCJMaySelfSuspend(false)
```

```
@SCJPhase({
    javax.safetycritical.annotate.Phase.STARTUP,
    javax.safetycritical.annotate.Phase.INITIALIZATION,
    javax.safetycritical.annotate.Phase.RUN,
    javax.safetycritical.annotate.Phase.CLEANUP })
public static javax.realtime.DeregistrationException get( )
```

Get the preallocated version of this Throwable. Allocation is done in memory that acts like **javax.realtime.ImmortalMemory** . The message and cause are cleared and the stack trace is filled out.

returns the preallocated exception

D.3.12 CLASS EnclosedType

```
@SCJAllowed
public final enum EnclosedType
```

Represents type size classes for deciding how large a lambda is. This size is dependent on what variables the lambda expression contains in its closure, i.e., it encloses. It is used by the {@code reserveLambda} methods in **javax.realtime.SizeEstimator** .

D.3.13 CLASS FirstInFirstOutScheduler

```
@SCJAllowed(javax.safetycritical.annotate.Level.LEVEL_1)
public class FirstInFirstOutScheduler extends javax.realtime.PriorityScheduler
```

A version of **javax.realtime.PriorityScheduler** where once a thread is scheduled at a given priority, it runs until it is blocked or is preempted by a higher priority thread. When preempted, it remains the next thread ready for its priority. This is the default scheduler for realtime tasks. It represents the required (by the RTSJ) priority-based scheduler. The default instance is the base scheduler which does fixed priority, preemptive scheduling.

Methods

```
@SCJAllowed
@SCJMayAllocate({})
@SCJMaySelfSuspend(false)
@Override
@SCJPhase({
    javax.safetycritical.annotate.Phase.STARTUP,
```

```
    javax.safetycritical.annotate.Phase.INITIALIZATION,
    javax.safetycritical.annotate.Phase.RUN,
    javax.safetycritical.annotate.Phase.CLEANUP })
public int getMaxPriority( )
```

Obtain the maximum priority available for a schedulable managed by this scheduler.

returns The value of the maximum priority.

```
@SCJAllowed
@SCJMayAllocate({})
@SCJMaySelfSuspend(false)
@Override
@SCJPhase({
    javax.safetycritical.annotate.Phase.STARTUP,
    javax.safetycritical.annotate.Phase.INITIALIZATION,
    javax.safetycritical.annotate.Phase.RUN,
    javax.safetycritical.annotate.Phase.CLEANUP })
public int getMinPriority( )
```

Obtain the minimum priority available for a schedulable managed by this scheduler.

returns The minimum priority used by this scheduler.

```
@SCJAllowed
@SCJMayAllocate({})
@SCJMaySelfSuspend(false)
@Override
@SCJPhase({
    javax.safetycritical.annotate.Phase.STARTUP,
    javax.safetycritical.annotate.Phase.INITIALIZATION,
    javax.safetycritical.annotate.Phase.RUN,
    javax.safetycritical.annotate.Phase.CLEANUP })
public int getNormPriority( )
```

Obtain the normal priority available for a schedulable managed by this scheduler.

returns The value of the normal priority.

```
@SCJAllowed(javax.safetycritical.annotate.Level.LEVEL_1)
@SCJMayAllocate({})
@SCJMaySelfSuspend(false)
@SCJPhase({
```

```
    javax.safetycritical.annotate.Phase.STARTUP,
    javax.safetycritical.annotate.Phase.INITIALIZATION,
    javax.safetycritical.annotate.Phase.RUN,
    javax.safetycritical.annotate.Phase.CLEANUP })
public static javax.realtime.FirstInFirstOutScheduler instance( )
```

Obtain a reference to the distinguished instance of {@code PriorityScheduler} which is the system's base scheduler.

returns A reference to the distinguished instance {@code PriorityScheduler}.

D.3.14 CLASS HighResolutionTime

```
@SCJAllowed
public abstract class HighResolutionTime<T extends HighResolutionTime<T>>
    implements java.lang.Comparable<T>, java.lang.Cloneable extends
    java.lang.Object
```

Class HighResolutionTime is the abstract base class for AbsoluteTime and RelativeTime, and is used to express time with nanosecond accuracy. When an API is defined that has an HighResolutionTime as a parameter, it can take either an absolute or relative time and will do something appropriate.

A time object in normalized form represents negative time if both components are nonzero and negative, or one is nonzero and negative and the other is zero. For add and subtract negative values behave as they do in arithmetic.

Caution: This class is explicitly unsafe in multithreaded situations when it is being changed. No synchronization is done. It is assumed that users of this class who are mutating instances will be doing their own synchronization at a higher level.

Methods

```
@SCJAllowed
@SCJMaySelfSuspend(false)
@SCJMayAllocate({})
@SCJPhase({
    javax.safetycritical.annotate.Phase.STARTUP,
    javax.safetycritical.annotate.Phase.INITIALIZATION,
    javax.safetycritical.annotate.Phase.RUN,
    javax.safetycritical.annotate.Phase.CLEANUP })
public int compareTo(T time)
```

Compares thisHighResolutionTime with the specified HighResolutionTime time.

time — Compares with the time of this.

Throws ClassCastException if the time parameter is not of the same class as this.

Throws IllegalArgumentException if the time parameter is not associated with the same clock as this, or when the time parameter is null.

returns a negative integer, zero, or a positive integer as this object is less than, equal to, or greater than time.

```
@SCJAllowed
@SCJMaySelfSuspend(false)
@SCJMayAllocate({})
@SCJPhase({
    javax.safetycritical.annotate.Phase.STARTUP,
    javax.safetycritical.annotate.Phase.INITIALIZATION,
    javax.safetycritical.annotate.Phase.RUN,
    javax.safetycritical.annotate.Phase.CLEANUP })
public boolean equals(T time)
```

Returns true if the argument time has the same type and values as this.

Equality includes clock association.

time — Value compared to this.

returns true if the parameter time is of the same type and has the same values as this.

```
@SCJAllowed
@SCJMaySelfSuspend(false)
@SCJMayAllocate({})
@SCJPhase({
    javax.safetycritical.annotate.Phase.STARTUP,
    javax.safetycritical.annotate.Phase.INITIALIZATION,
    javax.safetycritical.annotate.Phase.RUN,
    javax.safetycritical.annotate.Phase.CLEANUP })
public boolean equals(Object object)
```

Returns true if the argument object has the same type and values as this.

Equality includes clock association.

object — Value compared to this.

returns true if the parameter object is of the same type and has the same values as this.

```
@SCJAllowed
@SCJMaySelfSuspend(false)
@SCJMayAllocate({})
@SCJPhase({
    javax.safetycritical.annotate.Phase.STARTUP,
    javax.safetycritical.annotate.Phase.INITIALIZATION,
    javax.safetycritical.annotate.Phase.RUN,
    javax.safetycritical.annotate.Phase.CLEANUP })
public javax.realtime.Chronograph getChronograph( )
```

Get a reference to the **javax.realtime.Chronograph** associated with this.

returns a reference to the **javax.realtime.Chronograph** associated with this.

```
@SCJAllowed
@SCJMaySelfSuspend(false)
@SCJMayAllocate({})
@SCJPhase({
    javax.safetycritical.annotate.Phase.STARTUP,
    javax.safetycritical.annotate.Phase.INITIALIZATION,
    javax.safetycritical.annotate.Phase.RUN,
    javax.safetycritical.annotate.Phase.CLEANUP })
public javax.realtime.Clock getClock( )
```

Returns a reference to the clock associated with this.

returns a reference to the clock associated with this.

Throws UnsupportedOperationException if the time is based on a **javax.realtime.Chronograph** that is not a **javax.realtime.Clock** .

```
@SCJAllowed
@SCJMaySelfSuspend(false)
@SCJMayAllocate({})
@SCJPhase({
    javax.safetycritical.annotate.Phase.STARTUP,
    javax.safetycritical.annotate.Phase.INITIALIZATION,
    javax.safetycritical.annotate.Phase.RUN,
    javax.safetycritical.annotate.Phase.CLEANUP })
public final long getMilliseconds( )
```

Returns the milliseconds component of this.

returns the milliseconds component of the time represented by this.

```
@SCJAllowed
@SCJMaySelfSuspend(false)
@SCJMayAllocate({})
@SCJPhase({
    javax.safetycritical.annotate.Phase.STARTUP,
    javax.safetycritical.annotate.Phase.INITIALIZATION,
    javax.safetycritical.annotate.Phase.RUN,
    javax.safetycritical.annotate.Phase.CLEANUP })
public final int getNanoseconds( )
```

Returns the nanoseconds component of this.

returns the nanoseconds component of the time represented by this.

```
@SCJAllowed
@SCJMaySelfSuspend(false)
@SCJMayAllocate({})
@SCJPhase({
    javax.safetycritical.annotate.Phase.STARTUP,
    javax.safetycritical.annotate.Phase.INITIALIZATION,
    javax.safetycritical.annotate.Phase.RUN,
    javax.safetycritical.annotate.Phase.CLEANUP })
public int hashCode( )
```

Returns a hash code for this object in accordance with the general contract of hashCode. Time objects that are equals(HighResolutionTime) have the same hash code.

returns the hashcode value for this instance.

```
@SCJAllowed
@SCJMaySelfSuspend(false)
@SCJMayAllocate({})
@SCJPhase({
    javax.safetycritical.annotate.Phase.STARTUP,
    javax.safetycritical.annotate.Phase.INITIALIZATION,
    javax.safetycritical.annotate.Phase.RUN,
    javax.safetycritical.annotate.Phase.CLEANUP })
public T set(long millis)
```

Sets the millisecond component of this to the given argument, and the nanosecond component of this to 0. This method is equivalent to set(millis, 0).

millis — The desired value of the millisecond component of this.

```
@SCJAllowed
@SCJMaySelfSuspend(false)
@SCJMayAllocate({})
@SCJPhase({
   javax.safetycritical.annotate.Phase.STARTUP,
   javax.safetycritical.annotate.Phase.INITIALIZATION,
   javax.safetycritical.annotate.Phase.RUN,
   javax.safetycritical.annotate.Phase.CLEANUP })
public T set(long millis, int nanos)
```

Sets the millisecond and nanosecond components of this. The setting is subject to parameter normalization. If there is an overflow in the millisecond component while normalizing then an IllegalArgumentException will be thrown.

millis — The desired value for the millisecond component of this before normalization.

nanos — The desired value for the nanosecond component of this before normalization.

Throws IllegalArgumentException if there is an overflow in the millisecond component while normalizing.

```
@SCJAllowed
@SCJMaySelfSuspend(false)
@SCJMayAllocate({})
@SCJPhase({
   javax.safetycritical.annotate.Phase.STARTUP,
   javax.safetycritical.annotate.Phase.INITIALIZATION,
   javax.safetycritical.annotate.Phase.RUN,
   javax.safetycritical.annotate.Phase.CLEANUP })
public T set(T time)
```

Change the value represented by this to that of the given time. If the time parameter is null this method will throw IllegalArgumentException. If the type of this and the type of the given time are not the same, this method will throw ClassCastException. The clock associated with this is set to be the clock associated with the time parameter.

time — The new value for this.

Throws IllegalArgumentException if the parameter time is null.

Throws ClassCastException if the type of this and the type of the parameter time are not the same.

```
@SCJAllowed(javax.safetycritical.annotate.Level.LEVEL_2)
@SCJMayAllocate({})
@SCJMaySelfSuspend(true)
@SCJPhase({
    javax.safetycritical.annotate.Phase.STARTUP,
    javax.safetycritical.annotate.Phase.INITIALIZATION,
    javax.safetycritical.annotate.Phase.RUN,
    javax.safetycritical.annotate.Phase.CLEANUP })
public static boolean waitForObject(Object target, HighResolutionTime<?> time)
  throws java.lang.InterruptedException
```

Behaves exactly like target.wait() but with the enhancement that it waits with a precision of HighResolutionTime.

The wait time may be relative or absolute, and it is controlled by the clock associated with it. If the wait time is relative, then the calling thread is blocked waiting on target for the amount of time given by time, and measured by the associated clock. If the wait time is absolute, then the calling thread is blocked waiting on target until the indicated time value is reached by the associated clock.

target — The object on which to wait. The current thread must have a lock on the object.

time — The time for which to wait. If it is RelativeTime(0,0) then wait indefinitely. If it is null then wait indefinitely.

returns true if a notify was received before the timeout, False otherwise.

Throws InterruptedException if this schedulable object is interrupted by RealtimeThread.interrupt.

Throws IllegalArgumentException if time represents a relative time less than zero.

Throws IllegalMonitorStateException if target is not locked by the caller.

Throws UnsupportedOperationException if the wait operation is not supported using the clock associated with time.

See Also: java.lang.Object.wait(), java.lang.Object.wait(long), java.lang.Object.wait(long,int)

D.3.15 CLASS IllegalAssignmentError

```
@SCJAllowed
public class IllegalAssignmentError implements java.io.Serializable extends
    javax.realtime.StaticError
```

The exception thrown on an attempt to make an illegal assignment. For example, this will be thrown on any attempt to assign a reference to an object in scoped memory to a field of an object in immortal memory.

Methods

```
@SCJMayAllocate({javax.safetycritical.annotate.AllocationContext.CURRENT})
@SCJAllowed
@SCJMaySelfSuspend(false)
@SCJPhase({
   javax.safetycritical.annotate.Phase.STARTUP,
   javax.safetycritical.annotate.Phase.INITIALIZATION,
   javax.safetycritical.annotate.Phase.RUN,
   javax.safetycritical.annotate.Phase.CLEANUP })
public static javax.realtime.IllegalAssignmentError get( )
```

Obtain the singleton of this static throwable. It is prepared for immediate throwing.

returns the single instance of this throwable.

D.3.16 CLASS IllegalSchedulableStateException

```
@SCJAllowed
public class IllegalSchedulableStateException implements
    javax.realtime.StaticThrowable extends java.lang.IllegalThreadStateException
```

The exception thrown when a **javax.realtime.Schedulable** instance attempts an operation which is illegal in its current state. For instance, changing parameters on such instances are only allowed when the scheduler is not active or the new parameters are consistent with the current scheduler.

Methods

```
@SCJMayAllocate({})
@SCJAllowed
@SCJMaySelfSuspend(false)
@Override
@SCJPhase({
   javax.safetycritical.annotate.Phase.STARTUP,
   javax.safetycritical.annotate.Phase.INITIALIZATION,
   javax.safetycritical.annotate.Phase.RUN,
   javax.safetycritical.annotate.Phase.CLEANUP })
public java.lang.Throwable fillInStackTrace( )
```

Calls the infrastructure to capture the current stack trace in the schedulable object's local memory.

returns a reference to this Throwable.

```
@SCJMayAllocate({})
@SCJAllowed
@SCJMaySelfSuspend(false)
@SCJPhase({
   javax.safetycritical.annotate.Phase.STARTUP,
   javax.safetycritical.annotate.Phase.INITIALIZATION,
   javax.safetycritical.annotate.Phase.RUN,
   javax.safetycritical.annotate.Phase.CLEANUP })
public static javax.realtime.IllegalSchedulableStateException get( )
```

Get the preallocated version of this Throwable. Allocation is done in memory that acts like **javax.realtime.ImmortalMemory** . The message and cause are cleared and the stack trace is filled out.

returns the preallocated exception

```
@SCJMayAllocate({})
@SCJAllowed
@SCJMaySelfSuspend(false)
@Override
@SCJPhase({
   javax.safetycritical.annotate.Phase.STARTUP,
   javax.safetycritical.annotate.Phase.INITIALIZATION,
   javax.safetycritical.annotate.Phase.RUN,
   javax.safetycritical.annotate.Phase.CLEANUP })
public java.lang.Throwable getCause( )
```

getCause returns the cause of this exception or null when no cause was set by initCause. The cause is another exception that was caught just before this exception was thrown.

returns The cause or null.

```
@SCJMayAllocate({})
@SCJAllowed
@SCJMaySelfSuspend(false)
@Override
@SCJPhase({
   javax.safetycritical.annotate.Phase.STARTUP,
   javax.safetycritical.annotate.Phase.INITIALIZATION,
   javax.safetycritical.annotate.Phase.RUN,
   javax.safetycritical.annotate.Phase.CLEANUP })
public java.lang.String getLocalizedMessage( )
```

Subclasses may override this message to get an error message that is localized to the default locale.

By default it returns getMessage().

returns the error message

```
@SCJMayAllocate({})
@SCJAllowed
@SCJMaySelfSuspend(false)
@Override
@SCJPhase({
    javax.safetycritical.annotate.Phase.STARTUP,
    javax.safetycritical.annotate.Phase.INITIALIZATION,
    javax.safetycritical.annotate.Phase.RUN,
    javax.safetycritical.annotate.Phase.CLEANUP })
public java.lang.String getMessage( )
```

get the message describing the problem from the schedulable object's local memory.

returns the message given to the constructor or null when no message was set.

```
@SCJMayAllocate({})
@SCJAllowed
@SCJMaySelfSuspend(false)
@Override
@SCJPhase({
    javax.safetycritical.annotate.Phase.STARTUP,
    javax.safetycritical.annotate.Phase.INITIALIZATION,
    javax.safetycritical.annotate.Phase.RUN,
    javax.safetycritical.annotate.Phase.CLEANUP })
public java.lang.StackTraceElement[] getStackTrace( )
```

Get the stack trace created by fillInStackTrace for this Throwable as an array of StackTraceElements.

The stack trace does not need to contain entries for all methods that are actually on the call stack, the infrastructure may decide to skip some stack trace entries. Even an empty array is a valid result of this function.

Repeated calls of this function without intervening calls to fillInStackTrace will return the same result.

When memory areas are used, and this Throwable was allocated in a different memory area than the current allocation context, the resulting stack trace will be allocated in either the same memory area this was allocated in or the current memory area, depending on which is the least deeply nested, thereby creating objects that are assignment compatible with both areas.

returns array representing the stack trace, it is never null.

```
@SCJMayAllocate({})
@SCJAllowed
@SCJMaySelfSuspend(false)
@Override
@SCJPhase({
    javax.safetycritical.annotate.Phase.STARTUP,
    javax.safetycritical.annotate.Phase.INITIALIZATION,
    javax.safetycritical.annotate.Phase.RUN,
    javax.safetycritical.annotate.Phase.CLEANUP })
public java.lang.Throwable initCause(Throwable causingThrowable)
```

Initializes the cause to the given Throwable in the schedulable object's local memory.

causingThrowable — the reason why this Throwable gets Thrown.

returns the reference to this Throwable.

Throws IllegalArgumentException when the cause is this Throwable itself.

```
@SCJMayAllocate({})
@SCJAllowed
@SCJMaySelfSuspend(false)
@Override
@SCJPhase({
    javax.safetycritical.annotate.Phase.STARTUP,
    javax.safetycritical.annotate.Phase.INITIALIZATION,
    javax.safetycritical.annotate.Phase.RUN,
    javax.safetycritical.annotate.Phase.CLEANUP })
public java.lang.Throwable initMessage(String message)
```

Set the message in the schedulable object's local storage. This is the only method that is not also defined in Throwable.

message — is the text to set.

```
@SCJMayAllocate({})
@SCJAllowed
@SCJMaySelfSuspend(false)
@Override
@SCJPhase({
    javax.safetycritical.annotate.Phase.STARTUP,
    javax.safetycritical.annotate.Phase.INITIALIZATION,
    javax.safetycritical.annotate.Phase.RUN,
    javax.safetycritical.annotate.Phase.CLEANUP })
public void printStackTrace(PrintStream stream)
```

Print stack trace of this Throwable to stream.

The printed stack trace contains the result of toString() as the first line followed by one line for each stack trace element that contains the name of the method or constructor, optionally followed by the source file name and source file line number when available.

```
@SCJMayAllocate({})
@SCJAllowed
@SCJMaySelfSuspend(false)
@Override
@SCJPhase({
    javax.safetycritical.annotate.Phase.STARTUP,
    javax.safetycritical.annotate.Phase.INITIALIZATION,
    javax.safetycritical.annotate.Phase.RUN,
    javax.safetycritical.annotate.Phase.CLEANUP })
public void printStackTrace( )
```

Print stack trace of this Throwable to System.err.

The printed stack trace contains the result of toString() as the first line followed by one line for each stack trace element that contains the name of the method or constructor, optionally followed by the source file name and source file line number when available.

```
@SCJMayAllocate({})
@SCJAllowed
@SCJMaySelfSuspend(false)
@Override
@SCJPhase({
    javax.safetycritical.annotate.Phase.STARTUP,
    javax.safetycritical.annotate.Phase.INITIALIZATION,
    javax.safetycritical.annotate.Phase.RUN,
    javax.safetycritical.annotate.Phase.CLEANUP })
public void setStackTrace(StackTraceElement [] new_stackTrace)
  throws java.lang.NullPointerException
```

This method allows overriding the stack trace that was filled during construction of this object. It is intended to be used in a serialization context when the stack trace of a remote exception should be treated like a local.

new_stackTrace — the stack trace to replace be used.

Throws NullPointerException when new_stackTrace or any element of new_stackTrace is null.

D.3.17 CLASS ImmortalMemory

@SCJAllowed
public final class ImmortalMemory **extends** javax.realtime.MemoryArea

This class represents immortal memory. Objects allocated in immortal memory are never reclaimed during the lifetime of the application. The singleton instance of this class is created and managed by the infrastructure, so no application visible constructors or methods are provided.

D.3.18 CLASS InaccessibleAreaException

@SCJAllowed
public class InaccessibleAreaException **implements** java.io.Serializable **extends** javax.realtime.StaticRuntimeException

The exception thrown when a Schedulabe attempts to access a memory area that is not on the current schedulab's scope stack. TBD *** Andy: Is this possible in SCJ??

Methods

```
@SCJMayAllocate({})
@SCJAllowed
@SCJMaySelfSuspend(false)
@SCJPhase({
   javax.safetycritical.annotate.Phase.STARTUP,
   javax.safetycritical.annotate.Phase.INITIALIZATION,
   javax.safetycritical.annotate.Phase.RUN,
   javax.safetycritical.annotate.Phase.CLEANUP })
public static javax.realtime.InaccessibleAreaException get( )
```

Get the preallocated version of this Throwable. Allocation is done in memory that acts like **javax.realtime.ImmortalMemory** . The message and cause are cleared and the stack trace is filled out.

returns the preallocated exception

D.3.19 CLASS MemoryAccessError

@SCJAllowed
public class MemoryAccessError **extends** javax.realtime.StaticError

This error is thrown on an attempt to refer to an object in an inaccessible MemoryArea.

Constructors

```
@SCJAllowed
@SCJMaySelfSuspend(false)
@SCJMayAllocate({
   javax.safetycritical.annotate.AllocationContext.CURRENT,
   javax.safetycritical.annotate.AllocationContext.INNER,
   javax.safetycritical.annotate.AllocationContext.OUTER})
@SCJPhase({
   javax.safetycritical.annotate.Phase.STARTUP,
   javax.safetycritical.annotate.Phase.INITIALIZATION,
   javax.safetycritical.annotate.Phase.RUN,
   javax.safetycritical.annotate.Phase.CLEANUP })
public MemoryAccessError( )
```

A constructor for {@code MemoryAccessError}

```
@SCJAllowed
@SCJMaySelfSuspend(false)
@SCJMayAllocate({
   javax.safetycritical.annotate.AllocationContext.CURRENT,
   javax.safetycritical.annotate.AllocationContext.INNER,
   javax.safetycritical.annotate.AllocationContext.OUTER})
@SCJPhase({
   javax.safetycritical.annotate.Phase.STARTUP,
   javax.safetycritical.annotate.Phase.INITIALIZATION,
   javax.safetycritical.annotate.Phase.RUN,
   javax.safetycritical.annotate.Phase.CLEANUP })
public MemoryAccessError(String description)
```

D.3.20 CLASS MemoryArea

```
@SCJAllowed
public abstract class MemoryArea extends java.lang.Object
```

All allocation contexts are implemented by memory areas. This is the base class for all memory areas.

Methods

```
@SCJAllowed
@SCJMaySelfSuspend(false)
@SCJMayAllocate({
   javax.safetycritical.annotate.AllocationContext.CURRENT,
   javax.safetycritical.annotate.AllocationContext.INNER,
   javax.safetycritical.annotate.AllocationContext.OUTER})
```

```
@SCJPhase({
    javax.safetycritical.annotate.Phase.STARTUP,
    javax.safetycritical.annotate.Phase.INITIALIZATION,
    javax.safetycritical.annotate.Phase.RUN,
    javax.safetycritical.annotate.Phase.CLEANUP })
public static javax.realtime.MemoryArea getMemoryArea(Object object)
```

Get the memory area in which object is allocated,

returns the memory area

```
@SCJAllowed
@SCJMaySelfSuspend(false)
@SCJMayAllocate({
    javax.safetycritical.annotate.AllocationContext.CURRENT,
    javax.safetycritical.annotate.AllocationContext.INNER,
    javax.safetycritical.annotate.AllocationContext.OUTER})
@SCJPhase({
    javax.safetycritical.annotate.Phase.STARTUP,
    javax.safetycritical.annotate.Phase.INITIALIZATION,
    javax.safetycritical.annotate.Phase.RUN,
    javax.safetycritical.annotate.Phase.CLEANUP })
public boolean mayHoldReferenceTo(Object value)
```

Determine whether an object allocated in the memory area represented by this can hold a reference to the object value.

returns true when value can be assigned to a field of an object in this memory area, otherwise false.

```
@SCJAllowed
@SCJMaySelfSuspend(false)
@SCJMayAllocate({
    javax.safetycritical.annotate.AllocationContext.CURRENT,
    javax.safetycritical.annotate.AllocationContext.INNER,
    javax.safetycritical.annotate.AllocationContext.OUTER})
@SCJPhase({
    javax.safetycritical.annotate.Phase.STARTUP,
    javax.safetycritical.annotate.Phase.INITIALIZATION,
    javax.safetycritical.annotate.Phase.RUN,
    javax.safetycritical.annotate.Phase.CLEANUP })
public boolean mayHoldReferenceTo( )
```

Determine whether an object allocated in the memory area represented by this can hold a reference to an object allocated in the current memory area.

returns true when an object in the current memory area can be assigned to a field of an object in this memory area, otherwise false.

```
@SCJAllowed
@SCJMaySelfSuspend(false)
@SCJMayAllocate({
   javax.safetycritical.annotate.AllocationContext.CURRENT,
   javax.safetycritical.annotate.AllocationContext.INNER,
   javax.safetycritical.annotate.AllocationContext.OUTER})
@SCJPhase({
   javax.safetycritical.annotate.Phase.STARTUP,
   javax.safetycritical.annotate.Phase.INITIALIZATION,
   javax.safetycritical.annotate.Phase.RUN,
   javax.safetycritical.annotate.Phase.CLEANUP })
public long size( )
```

The size of a memory area is memoryConsumed() + memoryRemaining().

returns the total size of this memory area.

D.3.21 CLASS **MemoryInUseException**

```
@SCJAllowed
public class MemoryInUseException extends
    javax.realtime.StaticRuntimeException
```

The exception thrown when there has been attempt to allocate a range of physical or virtual memory that is already in use.

Methods

```
@SCJMayAllocate({})
@SCJAllowed
@SCJMaySelfSuspend(false)
@SCJPhase({
   javax.safetycritical.annotate.Phase.STARTUP,
   javax.safetycritical.annotate.Phase.INITIALIZATION,
   javax.safetycritical.annotate.Phase.RUN,
   javax.safetycritical.annotate.Phase.CLEANUP })
public static javax.realtime.MemoryInUseException get( )
```

Get the preallocated version of this Throwable. Allocation is done in memory that acts like **javax.realtime.ImmortalMemory** . The message and cause are cleared and the stack trace is filled out.

returns the preallocated exception

D.3.22 CLASS MemoryParameters

```
@SCJAllowed
public class MemoryParameters implements java.lang.Cloneable,
    java.io.Serializable extends java.lang.Object
```

This class is used to define the maximum amount of memory that a schedulable object requires in its default memory area (its per-release private scope memory) and in immortal memory. The SCJ restricts this class relative to the RTSJ such that values can be created but not queried or changed.

Fields

```
@SCJAllowed
public static final long UNLIMITED
```

```
@SCJAllowed
public static final long UNREFERENCED
```

Constructors

```
@SCJAllowed
@SCJMaySelfSuspend(false)
@SCJMayAllocate({})
@SCJPhase({
   javax.safetycritical.annotate.Phase.STARTUP,
   javax.safetycritical.annotate.Phase.INITIALIZATION,
   javax.safetycritical.annotate.Phase.RUN,
   javax.safetycritical.annotate.Phase.CLEANUP })
public MemoryParameters(long maxInitialArea, long maxImmortal)
```

Create a MemoryParameters object with the given maximum values.

maxInitialArea — is the maximum amount of memory in the per-release private memory area.

maxImmortal — is the maximum amount of memory in the immortal memory area required by the associated schedulable object.

Throws IllegalArgumentException if any value is negative, or if NO_MAX is passed as the value of maxMemoryArea or maxImmortal.

Methods

```
@SCJAllowed
@Override
```

```
@SCJMaySelfSuspend(false)
@SCJMayAllocate({javax.safetycritical.annotate.AllocationContext.CURRENT})
@SCJPhase({
   javax.safetycritical.annotate.Phase.STARTUP,
   javax.safetycritical.annotate.Phase.INITIALIZATION,
   javax.safetycritical.annotate.Phase.RUN,
   javax.safetycritical.annotate.Phase.CLEANUP })
public java.lang.Object clone( )
```

Create a clone of this MemoryParameters object.

D.3.23 CLASS MemoryTypeConflictException

```
@SCJAllowed
public class MemoryTypeConflictException extends
    javax.realtime.StaticRuntimeException
```

This exception is thrown when the PhysicalMemoryManager is given conflicting specifications for memory. The conflict can be between types in an array of memory type specifiers, or between the specifiers and a specified base address.

Methods

```
@SCJMayAllocate({})
@SCJAllowed
@SCJMaySelfSuspend(false)
@SCJPhase({
   javax.safetycritical.annotate.Phase.STARTUP,
   javax.safetycritical.annotate.Phase.INITIALIZATION,
   javax.safetycritical.annotate.Phase.RUN,
   javax.safetycritical.annotate.Phase.CLEANUP })
public static javax.realtime.MemoryTypeConflictException get( )
```

Get the preallocated version of this Throwable. Allocation is done in memory that acts like **javax.realtime.ImmortalMemory** . The message and cause are cleared and the stack trace is filled out.

returns the preallocated exception

D.3.24 CLASS OffsetOutOfBoundsException

```
@SCJAllowed
public class OffsetOutOfBoundsException extends
    javax.realtime.StaticRuntimeException
```

when the constructor of an RawMemoryAccess is given an invalid address.

Methods

```
@SCJMayAllocate({})
@SCJAllowed
@SCJMaySelfSuspend(false)
@SCJPhase({
   javax.safetycritical.annotate.Phase.STARTUP,
   javax.safetycritical.annotate.Phase.INITIALIZATION,
   javax.safetycritical.annotate.Phase.RUN,
   javax.safetycritical.annotate.Phase.CLEANUP })
public static javax.realtime.OffsetOutOfBoundsException get( )
```

Get the preallocated version of this Throwable. Allocation is done in memory that acts like **javax.realtime.ImmortalMemory** . The message and cause are cleared and the stack trace is filled out.

returns the preallocated exception

D.3.25 CLASS PeriodicParameters

```
@SCJAllowed
public class PeriodicParameters extends javax.realtime.ReleaseParameters
```

This RTSJ class is restricted so that it allows the start time and the period to be set but not to be subsequently changed or queried.

Constructors

```
@SCJAllowed
@SCJMaySelfSuspend(false)
@SCJMayAllocate({})
@SCJPhase({
   javax.safetycritical.annotate.Phase.STARTUP,
   javax.safetycritical.annotate.Phase.INITIALIZATION,
   javax.safetycritical.annotate.Phase.RUN,
   javax.safetycritical.annotate.Phase.CLEANUP })
public PeriodicParameters(RelativeTime start,
  RelativeTime period,
  RelativeTime deadline,
  AsyncEventHandler missHandler)
```

Construct a new PeriodicParameters object within the current memory area.

start — is time of the first release of the associated schedulable object relative to the start of the mission. A null value defaults to an offset of zero milliseconds.

period — is the time between each release of the associated schedulable object.

deadline — is an offset from the release time by which the release should finish. A null deadline indicates the same value as the period.

missHandler — is the AperiodicEventHandler to be released if the associated schedulable object misses its deadline. A null parameter indicates that no handler should be released.

Throws IllegalArgumentException if the period is null or its time value is not greater than zero, or if the time value of deadline is not greater than zero, or if the clock associated with the start, period and deadline parameters is not the same.

```
@SCJAllowed
@SCJMaySelfSuspend(false)
@SCJMayAllocate({})
@SCJPhase({
    javax.safetycritical.annotate.Phase.STARTUP,
    javax.safetycritical.annotate.Phase.INITIALIZATION,
    javax.safetycritical.annotate.Phase.RUN,
    javax.safetycritical.annotate.Phase.CLEANUP })
public PeriodicParameters(RelativeTime start, RelativeTime period)
```

This constructor behaves the same as calling PeriodicParameters(start, period, null, null).

Methods

```
@SCJAllowed
@Override
@SCJMaySelfSuspend(false)
@SCJMayAllocate({javax.safetycritical.annotate.AllocationContext.CURRENT})
@SCJPhase({
    javax.safetycritical.annotate.Phase.STARTUP,
    javax.safetycritical.annotate.Phase.INITIALIZATION,
    javax.safetycritical.annotate.Phase.RUN,
    javax.safetycritical.annotate.Phase.CLEANUP })
public java.lang.Object clone( )
```

Create a clone of this PeriodicParameters object.

D.3.26 CLASS PriorityParameters

```
@SCJAllowed
public class PriorityParameters extends javax.realtime.SchedulingParameters
```

This class is restricted relative to the RTSJ so that it allows the priority to be created and queried, but not changed.

In SCJ the range of priorities is separated into software priorities and hardware priorities (see Section 4.7.5). Hardware priorities have higher values than software priorities. Schedulable objects can be assigned only software priorities. Ceiling priorities can be either software or hardware priorities.

Constructors

```
@SCJAllowed
@SCJMaySelfSuspend(false)
@SCJMayAllocate({})
@SCJPhase({
   javax.safetycritical.annotate.Phase.STARTUP,
   javax.safetycritical.annotate.Phase.INITIALIZATION,
   javax.safetycritical.annotate.Phase.RUN,
   javax.safetycritical.annotate.Phase.CLEANUP })
public PriorityParameters(int priority)
```

Create a PriorityParameters object specifying the given priority.

priority — is the integer value of the specified priority.

Throws IllegalArgumentException if priority is not in the range of supported priorities.

Methods

```
@SCJAllowed
@SCJMaySelfSuspend(false)
@SCJMayAllocate({})
@SCJPhase({
   javax.safetycritical.annotate.Phase.STARTUP,
   javax.safetycritical.annotate.Phase.INITIALIZATION,
   javax.safetycritical.annotate.Phase.RUN,
   javax.safetycritical.annotate.Phase.CLEANUP })
public int getPriority( )
```

returns the integer priority value that was specified at construction time.

D.3.27 CLASS PriorityScheduler

```
@SCJAllowed
public abstract class PriorityScheduler extends javax.realtime.Scheduler
```

Priority-based dispatching is supported at Level 1 and Level 2. The only access to the priority scheduler is for obtaining the minimum and maximum priority.

Methods

```
@SCJAllowed
@SCJMaySelfSuspend(false)
@SCJMayAllocate({})
@SCJPhase({
    javax.safetycritical.annotate.Phase.STARTUP,
    javax.safetycritical.annotate.Phase.INITIALIZATION,
    javax.safetycritical.annotate.Phase.RUN,
    javax.safetycritical.annotate.Phase.CLEANUP })
public abstract int getMaxPriority( )
```

Gets the maximum software real-time priority supported by this scheduler.

returns the maximum priority supported by this scheduler.

```
@SCJAllowed
@SCJMaySelfSuspend(false)
@SCJMayAllocate({})
@SCJPhase({
    javax.safetycritical.annotate.Phase.STARTUP,
    javax.safetycritical.annotate.Phase.INITIALIZATION,
    javax.safetycritical.annotate.Phase.RUN,
    javax.safetycritical.annotate.Phase.CLEANUP })
public abstract int getMinPriority( )
```

Gets the minimum software real-time priority supported by this scheduler.

returns the minimum priority supported by this scheduler.

```
@SCJAllowed
@SCJMaySelfSuspend(false)
@SCJMayAllocate({})
@SCJPhase({
    javax.safetycritical.annotate.Phase.STARTUP,
    javax.safetycritical.annotate.Phase.INITIALIZATION,
    javax.safetycritical.annotate.Phase.RUN,
    javax.safetycritical.annotate.Phase.CLEANUP })
public abstract int getNormPriority( )
```

returns the normal software real-time priority supported by this scheduler.

D.3.28 CLASS ProcessorAffinityException

```
@SCJAllowed(javax.safetycritical.annotate.Level.LEVEL_1)
public class ProcessorAffinityException extends
    javax.realtime.StaticCheckedException
```

Exception used to report processor affinity-related errors.

Methods

```
@SCJMayAllocate({})
@SCJAllowed
@SCJMaySelfSuspend(false)
@SCJPhase({
   javax.safetycritical.annotate.Phase.STARTUP,
   javax.safetycritical.annotate.Phase.INITIALIZATION,
   javax.safetycritical.annotate.Phase.RUN,
   javax.safetycritical.annotate.Phase.CLEANUP })
public static javax.realtime.ProcessorAffinityException get( )
```

Obtain the singleton of this static throwable. It is prepared for immediate throwing.

returns the single instance of this throwable.

D.3.29 CLASS RealtimeThread

```
@SCJAllowed
public class RealtimeThread implements javax.realtime.BoundSchedulable,
    javax.realtime.AsyncTimable extends java.lang.Thread
```

Real-time threads cannot be directly created by an SCJ application. However, they are needed by the infrastructure to support ManagedThreads.

The class declares some static methods that can be used by all managed schedulables. For example, the spin method can be used at Level 0, hence the class is visible at Level 0.

Methods

```
@SCJAllowed(javax.safetycritical.annotate.Level.SUPPORT)
@SCJMaySelfSuspend(false)
@SCJMayAllocate({})
@Override
@SCJPhase({
```

```
    javax.safetycritical.annotate.Phase.STARTUP,
    javax.safetycritical.annotate.Phase.INITIALIZATION,
    javax.safetycritical.annotate.Phase.RUN,
    javax.safetycritical.annotate.Phase.CLEANUP })
public final void fire( )
```

Used by the SCJ infrastructure to support the time release of real-time threads and timers with user-defined clocks. Should not be called by the application.

```
@SCJMaySelfSuspend(false)
@SCJPhase({
    javax.safetycritical.annotate.Phase.STARTUP,
    javax.safetycritical.annotate.Phase.INITIALIZATION,
    javax.safetycritical.annotate.Phase.RUN,
    javax.safetycritical.annotate.Phase.CLEANUP})
@SCJMayAllocate({})
@SCJAllowed(javax.safetycritical.annotate.Level.LEVEL_1)
@Override
public javax.realtime.Affinity getAffinity( )
```

```
@SCJMaySelfSuspend(false)
@SCJPhase({
    javax.safetycritical.annotate.Phase.STARTUP,
    javax.safetycritical.annotate.Phase.INITIALIZATION})
@SCJMayAllocate({})
@SCJAllowed(javax.safetycritical.annotate.Level.LEVEL_1)
@Override
public void setAffinity(Affinity set)
  throws java.lang.IllegalArgumentException,
      javax.realtime.ProcessorAffinityException, java.lang.NullPointerException
```

```
@SCJAllowed(javax.safetycritical.annotate.Level.LEVEL_2)
@SCJMaySelfSuspend(true)
@SCJMayAllocate({})
@SCJPhase({
    javax.safetycritical.annotate.Phase.INITIALIZATION,
    javax.safetycritical.annotate.Phase.RUN,
    javax.safetycritical.annotate.Phase.CLEANUP})
public static void sleep(HighResolutionTime<?> time)
  throws java.lang.InterruptedException
```

Removes the currently execution schedulable object from the set of runnable schedulable objects until time.

Throws InterruptedException when the thread is interrupted by interrupt() during the time between calling this method and returning from it. This exception cannot be thrown if the method is called from a managed event handler.

Throws IllegalArgumentException when time is null, when time is a relative time less than zero, or when the Chronograph of time is not a Clock.

```
@SCJAllowed
@SCJMaySelfSuspend(true)
@SCJMayAllocate({})
@SCJPhase({
    javax.safetycritical.annotate.Phase.STARTUP,
    javax.safetycritical.annotate.Phase.INITIALIZATION,
    javax.safetycritical.annotate.Phase.RUN,
    javax.safetycritical.annotate.Phase.CLEANUP })
public static void spin(int nanos)
  throws java.lang.InterruptedException, java.lang.ClassCastException,
      java.lang.IllegalArgumentException
```

The same as calling spin(HighResolutionTime) with a relative time on the default real-time clock,of zero milliseconds, and nanos.

nanos — the number of nanoseconds to wait.

Throws InterruptedException when the thread is interrupted by interrupt() during the time between calling this method and returning from it. This exception cannot be thrown if the method is called from a managed event handler.

Throws IllegalArgumentException when nanos is less than zero.

```
@SCJAllowed
@SCJMaySelfSuspend(false)
@SCJMayAllocate({})
@SCJPhase({
    javax.safetycritical.annotate.Phase.STARTUP,
    javax.safetycritical.annotate.Phase.INITIALIZATION,
    javax.safetycritical.annotate.Phase.RUN,
    javax.safetycritical.annotate.Phase.CLEANUP })
public static void spin(HighResolutionTime<?> time)
  throws java.lang.InterruptedException, java.lang.ClassCastException,
      java.lang.IllegalArgumentException
```

Similar to sleep(HighResolutionTime) except it performs a busy wait by polling the Chronograph for the duration of time.

time — an absolute or relative time at which to stop spinning.

Throws InterruptedException when the thread is interrupted by interrupt() during the time between calling this method and returning from it. This exception cannot be thrown if the method is called from a managed event handler.

Throws IllegalArgumentException when time is null , or when time is a relative time less than zero.

```
@SCJAllowed(javax.safetycritical.annotate.Level.LEVEL_2)
@SCJMaySelfSuspend(true)
@SCJMayAllocate({})
@SCJPhase({
    javax.safetycritical.annotate.Phase.INITIALIZATION,
    javax.safetycritical.annotate.Phase.RUN,
    javax.safetycritical.annotate.Phase.CLEANUP})
public static void suspend(HighResolutionTime<?> time)
```

The same as sleep(HighResolutionTime) except that it is not interruptible.

time — an absolute or relative time until which to suspend.

Throws IllegalArgumentException when time is null, when time is a relative time less than zero, or when the Chronograph of time is not a Clock.

D.3.30 CLASS **RegistrationException**

```
@SCJAllowed
public class RegistrationException extends
      javax.realtime.StaticRuntimeException
```

The exception thrown when registering an InterruptServiceRoutine

Methods

```
@SCJMayAllocate({})
@SCJAllowed
@SCJMaySelfSuspend(false)
@SCJPhase({
    javax.safetycritical.annotate.Phase.STARTUP,
    javax.safetycritical.annotate.Phase.INITIALIZATION,
    javax.safetycritical.annotate.Phase.RUN,
    javax.safetycritical.annotate.Phase.CLEANUP })
public static javax.realtime.RegistrationException get( )
```

Get the preallocated version of this Throwable. Allocation is done in memory that acts like **javax.realtime.ImmortalMemory** . The message and cause are cleared and the stack trace is filled out.

returns the preallocated exception

D.3.31 CLASS **RelativeTime**

```
@SCJAllowed
public class RelativeTime extends javax.realtime.HighResolutionTime
```

An object that represents a time interval represented by a number of milliseconds plus nanoseconds. The time interval is kept in normalized form.

A negative interval relative to now represents time in the past. For add and subtract negative values behave as they do in arithmetic.

Caution: This class is explicitly unsafe in multithreaded situations when it is being changed. No synchronization is done. It is assumed that users of this class who are mutating instances will be doing their own synchronization at a higher level.

Constructors

```
@SCJAllowed
@SCJMaySelfSuspend(false)
@SCJMayAllocate({})
@SCJPhase({
   javax.safetycritical.annotate.Phase.STARTUP,
   javax.safetycritical.annotate.Phase.INITIALIZATION,
   javax.safetycritical.annotate.Phase.RUN,
   javax.safetycritical.annotate.Phase.CLEANUP })
public RelativeTime(long millis, int nanos, Chronograph clock)
```

Construct a RelativeTime object representing an interval based on the parameter millis plus the parameter nanos. The construction is subject to millis and nanos parameters normalization. If there is an overflow in the millisecond component when normalizing then an IllegalArgumentException will be thrown.

The clock association is made with the clock parameter.

millis — The desired value for the millisecond component of this. The actual value is the result of parameter normalization.

nanos — The desired value for the nanosecond component of this. The actual value is the result of parameter normalization.

clock — The chronograph providing the association for the newly constructed object. If chronograph is null the association is made with the real-time clock.

Throws IllegalArgumentException if there is an overflow in the millisecond component when normalizing.

```
@SCJAllowed
@SCJMaySelfSuspend(false)
@SCJMayAllocate({})
@SCJPhase({
    javax.safetycritical.annotate.Phase.STARTUP,
    javax.safetycritical.annotate.Phase.INITIALIZATION,
    javax.safetycritical.annotate.Phase.RUN,
    javax.safetycritical.annotate.Phase.CLEANUP })
public RelativeTime( )
```

This constructor behaves the same as calling RelativeTime(0, 0, null).

```
@SCJAllowed
@SCJMaySelfSuspend(false)
@SCJMayAllocate({})
@SCJPhase({
    javax.safetycritical.annotate.Phase.STARTUP,
    javax.safetycritical.annotate.Phase.INITIALIZATION,
    javax.safetycritical.annotate.Phase.RUN,
    javax.safetycritical.annotate.Phase.CLEANUP })
public RelativeTime(long millis, int nanos)
```

This constructor behaves the same as calling RelativeTime(millis, nanos, null).

```
@SCJAllowed
@SCJMaySelfSuspend(false)
@SCJMayAllocate({})
@SCJPhase({
    javax.safetycritical.annotate.Phase.STARTUP,
    javax.safetycritical.annotate.Phase.INITIALIZATION,
    javax.safetycritical.annotate.Phase.RUN,
    javax.safetycritical.annotate.Phase.CLEANUP })
public RelativeTime(Chronograph clock)
```

This constructor behaves the same as calling RelativeTime(0, 0, clock).

```
@SCJAllowed
@SCJMaySelfSuspend(false)
@SCJMayAllocate({javax.safetycritical.annotate.AllocationContext.CURRENT})
@SCJPhase({
    javax.safetycritical.annotate.Phase.STARTUP,
    javax.safetycritical.annotate.Phase.INITIALIZATION,
    javax.safetycritical.annotate.Phase.RUN,
    javax.safetycritical.annotate.Phase.CLEANUP })
public RelativeTime(RelativeTime time)
```

Make a new RelativeTime object from the given RelativeTime object.

The new object will have the same clock association as the time parameter.

time — The RelativeTime object which is the source for the copy.

Throws IllegalArgumentException if the time parameter is null.

Methods

```
@SCJAllowed
@SCJMaySelfSuspend(false)
@SCJMayAllocate({javax.safetycritical.annotate.AllocationContext.CURRENT})
@SCJPhase({
   javax.safetycritical.annotate.Phase.STARTUP,
   javax.safetycritical.annotate.Phase.INITIALIZATION,
   javax.safetycritical.annotate.Phase.RUN,
   javax.safetycritical.annotate.Phase.CLEANUP })
public javax.realtime.RelativeTime add(RelativeTime time)
```

Create a new instance of RelativeTime representing the result of adding time to the value of this and normalizing the result. The clock associated with this and the clock associated with the time parameter are expected to be the same, and such association is used for the result. An IllegalArgumentException is thrown if the clock associated with this and the clock associated with the time parameter are different. An IllegalArgumentException is thrown if the time parameter is null. An ArithmeticException is thrown if the result does not fit in the normalized format.

time — The time to add to this.

returns a new RelativeTime object whose time is the normalization of this plus the parameter time.

Throws IllegalArgumentException if the clock associated with this and the clock associated with the time parameter are different, or when the time parameter is null.

Throws ArithmeticException if the result does not fit in the normalized format.

```
@SCJAllowed
@SCJMaySelfSuspend(false)
@SCJMayAllocate({javax.safetycritical.annotate.AllocationContext.CURRENT})
@SCJPhase({
   javax.safetycritical.annotate.Phase.STARTUP,
   javax.safetycritical.annotate.Phase.INITIALIZATION,
   javax.safetycritical.annotate.Phase.RUN,
   javax.safetycritical.annotate.Phase.CLEANUP })
public javax.realtime.RelativeTime add(RelativeTime time, RelativeTime dest)
```

Return an object containing the value resulting from adding time to the value of this and normalizing the result. If dest is not null, the result is placed there and returned. Otherwise, a new object is allocated for the result. The clock associated with this and the clock associated with the time parameter are expected to be the same, and such association is used for the result. The clock associated with the dest parameter is ignored. An IllegalArgumentException is thrown if the clock associated with this and the clock associated with the time parameter are different. An IllegalArgumentException is thrown if the time parameter is null. An ArithmeticException is thrown if the result does not fit in the normalized format.

time — The time to add to this.

dest — If dest is not null, the result is placed there and returned. Otherwise, a new object is allocated for the result.

returns the value of dest if dest is not null, otherwise the result is returned in a newly allocated object.

Throws IllegalArgumentException if the clock associated with this and the clock associated with the time parameter are different, or when the time parameter is null.

Throws ArithmeticException if the result does not fit in the normalized format.

```
@SCJAllowed
@SCJMaySelfSuspend(false)
@SCJMayAllocate({javax.safetycritical.annotate.AllocationContext.CURRENT})
@SCJPhase({
   javax.safetycritical.annotate.Phase.STARTUP,
   javax.safetycritical.annotate.Phase.INITIALIZATION,
   javax.safetycritical.annotate.Phase.RUN,
   javax.safetycritical.annotate.Phase.CLEANUP })
public javax.realtime.RelativeTime add(long millis, int nanos, RelativeTime dest)
```

Return an object containing the value resulting from adding millis and nanos to the values from this and normalizing the result. If dest is not null, the result is placed there and returned. Otherwise, a new object is allocated for the result. The result will have the same clock association as this, and the clock association with dest is ignored. An ArithmeticException is thrown if the result does not fit in the normalized format.

millis — The number of milliseconds to be added to this.

nanos — The number of nanoseconds to be added to this.

dest — If dest is not null, the result is placed there and returned. Otherwise, a new object is allocated for the result.

returns the value of dest if dest is not null, otherwise a new object representing the result.

Throws ArithmeticException if the result does not fit in the normalized format.

```
@SCJAllowed
@SCJMaySelfSuspend(false)
@SCJMayAllocate({javax.safetycritical.annotate.AllocationContext.CURRENT})
@SCJPhase({
    javax.safetycritical.annotate.Phase.STARTUP,
    javax.safetycritical.annotate.Phase.INITIALIZATION,
    javax.safetycritical.annotate.Phase.RUN,
    javax.safetycritical.annotate.Phase.CLEANUP })
public javax.realtime.RelativeTime add(long millis, int nanos)
```

Create a new object representing the result of adding millis and nanos to the values from this and normalizing the result. The result will have the same clock association as this. An ArithmeticException is thrown if the result does not fit in the normalized format.

millis — The number of milliseconds to be added to this.

nanos — The number of nanoseconds to be added to this.

returns a new RelativeTime object whose time is the normalization of this plus millis and nanos.

Throws ArithmeticException if the result does not fit in the normalized format.

returns A new object containing the result of the addition.

```
@SCJAllowed
@SCJMaySelfSuspend(false)
@SCJMayAllocate({})
@Override
@SCJPhase({
    javax.safetycritical.annotate.Phase.STARTUP,
    javax.safetycritical.annotate.Phase.INITIALIZATION,
    javax.safetycritical.annotate.Phase.RUN,
    javax.safetycritical.annotate.Phase.CLEANUP })
public int compareTo(RelativeTime time)
```

Compares this object with the specified object for order.

returns a negative integer, zero, or a positive integer as this object is less than, equal to, or greater than the specified object.

```
@SCJAllowed
@SCJMaySelfSuspend(false)
@SCJMayAllocate({javax.safetycritical.annotate.AllocationContext.CURRENT})
@SCJPhase({
   javax.safetycritical.annotate.Phase.STARTUP,
   javax.safetycritical.annotate.Phase.INITIALIZATION,
   javax.safetycritical.annotate.Phase.RUN,
   javax.safetycritical.annotate.Phase.CLEANUP })
public javax.realtime.RelativeTime subtract(RelativeTime time, RelativeTime dest)
```

Return an object containing the value resulting from subtracting the value of time from the value of this and normalizing the result. If dest is not null, the result is placed there and returned. Otherwise, a new object is allocated for the result. The clock associated with this and the clock associated with the time parameter are expected to be the same, and such association is used for the result. The clock associated with the dest parameter is ignored. An IllegalArgumentException is thrown if the clock associated with this and the clock associated with the time parameter are different. An IllegalArgumentException is thrown if the time parameter is null. An ArithmeticException is thrown if the result does not fit in the normalized format.

time — The time to subtract from this.

dest — If dest is not null, the result is placed there and returned. Otherwise, a new object is allocated for the result.

returns the value of dest if dest is not null, otherwise a new object representing the result.

Throws IllegalArgumentException if the if the clock associated with this and the clock associated with the time parameter are different, or when the time parameter is null.

Throws ArithmeticException if the result does not fit in the normalized format.

```
@SCJAllowed
@SCJMaySelfSuspend(false)
@SCJMayAllocate({javax.safetycritical.annotate.AllocationContext.CURRENT})
@SCJPhase({
   javax.safetycritical.annotate.Phase.STARTUP,
   javax.safetycritical.annotate.Phase.INITIALIZATION,
   javax.safetycritical.annotate.Phase.RUN,
   javax.safetycritical.annotate.Phase.CLEANUP })
public javax.realtime.RelativeTime subtract(RelativeTime time)
```

Create a new instance of RelativeTime representing the result of subtracting time from the value of this and normalizing the result. The clock associated

with this and the clock associated with the time parameter are expected to be the same, and such association is used for the result. An IllegalArgumentException is thrown if the clock associated with this and the clock associated with the time parameter are different. An IllegalArgumentException is thrown if the time parameter is null. An ArithmeticException is thrown if the result does not fit in the normalized format.

time — The time to subtract from this.

returns a new RelativeTime object whose time is the normalization of this minus the parameter time parameter time.

Throws IllegalArgumentException if the clock associated with this and the clock associated with the time parameter are different, or when the time parameter is null.

Throws ArithmeticException if the result does not fit in the normalized format.

D.3.32 CLASS **ReleaseParameters**

@SCJAllowed
public abstract class ReleaseParameters **implements** java.lang.Cloneable, java.io.Serializable **extends** java.lang.Object

This is the base class for the release parameters hierarchy. All schedulability analysis of safety critical software is performed by the application developers offline. Although the RTSJ allows on-line schedulability analysis, SCJ assumes any such analysis is performed off line and that the on-line environment is predictable. Consequently, the assumption is that deadlines are not missed. However, to facilitate fault-tolerant applications, SCJ does support a deadline miss detection facility at Level 1 and Level 2. SCJ provides no direct mechanisms for coping with cost overruns.

The ReleaseParameters class hierarchy is restricted so that the parameters can be set, but not changed or queried.

D.3.33 CLASS **Scheduler**

@SCJAllowed
public abstract class Scheduler **extends** java.lang.Object

The RTSJ supported generic on-line feasibility analysis via the Scheduler class prior to RTSJ version 2.0, but this is now deprecated in version 2.0. SCJ supports only off-line schedulability analysis; hence all of the methods in this class are omitted.

Methods

```
@SCJAllowed
@SCJPhase({
    javax.safetycritical.annotate.Phase.INITIALIZATION,
    javax.safetycritical.annotate.Phase.RUN,
    javax.safetycritical.annotate.Phase.CLEANUP})
@SCJMaySelfSuspend(false)
@SCJMayAllocate({})
public static javax.realtime.Schedulable currentSchedulable( )
```

Gets the current schedulable.

returns a reference to the calling Schedulable.

Throws UnsupportedOperationException if called from an interrupt handler.

D.3.34 CLASS SchedulingParameters

```
@SCJAllowed
public abstract class SchedulingParameters implements java.lang.Cloneable,
    java.io.Serializable extends java.lang.Object
```

The RTSJ potentially allows different schedulers to be supported and defines this class as the root class for all scheduling parameters. In SCJ this class is empty; only priority parameters are supported.

There is no ImportanceParameters subclass in SCJ.

Methods

```
@SCJAllowed
@Override
@SCJMaySelfSuspend(false)
@SCJMayAllocate({javax.safetycritical.annotate.AllocationContext.CURRENT})
@SCJPhase({
    javax.safetycritical.annotate.Phase.STARTUP,
    javax.safetycritical.annotate.Phase.INITIALIZATION,
    javax.safetycritical.annotate.Phase.RUN,
    javax.safetycritical.annotate.Phase.CLEANUP })
public java.lang.Object clone( )
```

Create a clone of this SchedulingParameters object.

D.3.35 CLASS SizeEstimator

```
@SCJAllowed
public final class SizeEstimator extends java.lang.Object
```

This class maintains a conservative upper bound of the amount of memory required to store a set of objects.

Many objects allocate other objects when they are constructed. SizeEstimator only estimates the memory requirement of the object itself; it does not include memory required for any objects allocated at construction time. If the Java implementation allocates a single Java object in several parts not separately visible to the application (if, for example, the object and its monitor are separate), the size estimate shall include the sum of the sizes of all the invisible parts that are allocated from the same memory area as the object.

Alignment considerations, and possibly other order-dependent issues may cause the allocator to leave a small amount of unusable space. Consequently, the size estimate cannot be seen as more than a close estimate, but SCJ requires that the size estimate shall represent a tight upper bound.

Constructors

```
@SCJAllowed
@SCJPhase({
    javax.safetycritical.annotate.Phase.INITIALIZATION,
    javax.safetycritical.annotate.Phase.RUN,
    javax.safetycritical.annotate.Phase.CLEANUP})
@SCJMaySelfSuspend(false)
@SCJMayAllocate({
    javax.safetycritical.annotate.AllocationContext.CURRENT,
    javax.safetycritical.annotate.AllocationContext.INNER,
    javax.safetycritical.annotate.AllocationContext.OUTER})
public SizeEstimator( )
```

Creates a new SizeEstimator object in the current allocation context.

Methods

```
@SCJAllowed
@SCJPhase({
    javax.safetycritical.annotate.Phase.INITIALIZATION,
    javax.safetycritical.annotate.Phase.RUN,
    javax.safetycritical.annotate.Phase.CLEANUP})
@SCJMayAllocate({})
@SCJMaySelfSuspend(false)
public void clear( )
```

Return the estimate to zero for reuse.

```
@SCJAllowed
@SCJPhase({
   javax.safetycritical.annotate.Phase.INITIALIZATION,
   javax.safetycritical.annotate.Phase.RUN,
   javax.safetycritical.annotate.Phase.CLEANUP})
@SCJMayAllocate({})
@SCJMaySelfSuspend(false)
public long getEstimate( )
```

Gets an estimate of the number of bytes needed to store all the objects reserved.

returns the estimated size in bytes.

```
@SCJAllowed
@SCJPhase({
   javax.safetycritical.annotate.Phase.INITIALIZATION,
   javax.safetycritical.annotate.Phase.RUN,
   javax.safetycritical.annotate.Phase.CLEANUP})
@SCJMayAllocate({})
@SCJMaySelfSuspend(false)
public void reserve(SizeEstimator size, int num)
```

Adds num times the value returned by size.getEstimate to the currently computed size of the set of reserved objects.

size — is the size.SizeEstimator whose size is to be reserved.

num — is the number of times to reserve this amount.

Throws IllegalArgumentException if size is null or num is negative.

```
@SCJAllowed
@SCJPhase({
   javax.safetycritical.annotate.Phase.INITIALIZATION,
   javax.safetycritical.annotate.Phase.RUN,
   javax.safetycritical.annotate.Phase.CLEANUP})
@SCJMayAllocate({})
@SCJMaySelfSuspend(false)
public void reserve(SizeEstimator size)
```

Adds the value returned by size.getEstimate to the currently computed size of the set of reserved objects.

size — is the size.SizeEstimator whose size is to be reserved.

Throws IllegalArgumentException if size is null.

```
@SCJAllowed
@SCJPhase({
   javax.safetycritical.annotate.Phase.INITIALIZATION,
   javax.safetycritical.annotate.Phase.RUN,
   javax.safetycritical.annotate.Phase.CLEANUP})
@SCJMayAllocate({})
@SCJMaySelfSuspend(false)
public void reserve(Class<?> clss, int num)
```

Adds the required memory size of num instances of a clss object to the currently computed size of the set of reserved objects.

clss — is the class to take into account.

num — is the number of instances of clss to estimate.

Throws IllegalArgumentException if clss is null or num is negative.

```
@SCJAllowed
@SCJPhase({
   javax.safetycritical.annotate.Phase.INITIALIZATION,
   javax.safetycritical.annotate.Phase.RUN,
   javax.safetycritical.annotate.Phase.CLEANUP})
@SCJMayAllocate({})
@SCJMaySelfSuspend(false)
public void reserveArray(int length, Class<?> type)
```

Adds the required memory size of an additional instance of an array of length primitive values of Class type to the currently computed size of the set of reserved objects. Class values for the primitive types shall be chosen from primitive class types such as Integer.TYPE, and Float.TYPE. The reservation shall leave room for an array of length of the primitive type corresponding to type.

length — is the number of entries in the array.

type — is the class representing a primitive type.

Throws IllegalArgumentException if length is negative, or type does not represent a primitive type.

```
@SCJAllowed
@SCJPhase({
   javax.safetycritical.annotate.Phase.INITIALIZATION,
   javax.safetycritical.annotate.Phase.RUN,
   javax.safetycritical.annotate.Phase.CLEANUP})
@SCJMayAllocate({})
@SCJMaySelfSuspend(false)
public void reserveArray(int length)
```

Adds the size of an instance of an array of length reference values to the currently computed size of the set of reserved objects.

length — is the number of entries in the array.

Throws IllegalArgumentException if length is negative.

```
@SCJAllowed
@SCJPhase({
   javax.safetycritical.annotate.Phase.INITIALIZATION,
   javax.safetycritical.annotate.Phase.RUN,
   javax.safetycritical.annotate.Phase.CLEANUP})
@SCJMayAllocate({})
@SCJMaySelfSuspend(false)
public void reserveLambda( )
```

Determine the size of a lambda with no closure and add it to this size estimator.

```
@SCJAllowed
@SCJPhase({
   javax.safetycritical.annotate.Phase.INITIALIZATION,
   javax.safetycritical.annotate.Phase.RUN,
   javax.safetycritical.annotate.Phase.CLEANUP})
@SCJMayAllocate({})
@SCJMaySelfSuspend(false)
public void reserveLambda(EnclosedType first, EnclosedType second)
```

Determine the size of a lambda with two variables in its closure and add it to this size estimator.

first — Type of first variable in closure.

second — Type of second variable in closure.

```
@SCJAllowed
@SCJPhase({
   javax.safetycritical.annotate.Phase.INITIALIZATION,
   javax.safetycritical.annotate.Phase.RUN,
   javax.safetycritical.annotate.Phase.CLEANUP})
@SCJMayAllocate({})
@SCJMaySelfSuspend(false)
public void reserveLambda(EnclosedType first,
  EnclosedType second,
  EnclosedType [] others)
```

Determine the size of a lambda with more than two variables in its closure and add it to this size estimator.

first — Type of first variable in closure.

second — Type of second variable in closure.

others — Types of additional variables in closure.

D.3.36 CLASS SizeOutOfBoundsException

```
@SCJAllowed
public class SizeOutOfBoundsException extends
    javax.realtime.StaticRuntimeException
```

To throw when a memory access generated by a raw memory accessor instance (See RawMemory.) would cause access to an invalid address.

Methods

```
@SCJMayAllocate({})
@SCJAllowed
@SCJMaySelfSuspend(false)
@SCJPhase({
   javax.safetycritical.annotate.Phase.STARTUP,
   javax.safetycritical.annotate.Phase.INITIALIZATION,
   javax.safetycritical.annotate.Phase.RUN,
   javax.safetycritical.annotate.Phase.CLEANUP })
public static javax.realtime.SizeOutOfBoundsException get( )
```

Get the preallocated version of this Throwable. Allocation is done in memory that acts like **javax.realtime.ImmortalMemory** . The message and cause are cleared and the stack trace is filled out.

returns the preallocated exception

D.3.37 CLASS StaticError

```
@SCJAllowed
public abstract class StaticError implements javax.realtime.StaticThrowable
    extends java.lang.Error
```

Methods

```
@SCJMayAllocate({})
@SCJAllowed
@SCJMaySelfSuspend(false)
```

```
@Override
@SCJPhase({
    javax.safetycritical.annotate.Phase.STARTUP,
    javax.safetycritical.annotate.Phase.INITIALIZATION,
    javax.safetycritical.annotate.Phase.RUN,
    javax.safetycritical.annotate.Phase.CLEANUP })
public java.lang.Throwable fillInStackTrace( )
```

Calls the infrastructure to capture the current stack trace in the schedulable object's local memory.

returns a reference to this Throwable.

```
@SCJMayAllocate({})
@SCJAllowed
@SCJMaySelfSuspend(false)
@Override
@SCJPhase({
    javax.safetycritical.annotate.Phase.STARTUP,
    javax.safetycritical.annotate.Phase.INITIALIZATION,
    javax.safetycritical.annotate.Phase.RUN,
    javax.safetycritical.annotate.Phase.CLEANUP })
public java.lang.Throwable getCause( )
```

getCause returns the cause of this exception or null when no cause was set by initCause. The cause is another exception that was caught just before this exception was thrown.

returns The cause or null.

```
@SCJMayAllocate({})
@SCJAllowed
@SCJMaySelfSuspend(false)
@Override
@SCJPhase({
    javax.safetycritical.annotate.Phase.STARTUP,
    javax.safetycritical.annotate.Phase.INITIALIZATION,
    javax.safetycritical.annotate.Phase.RUN,
    javax.safetycritical.annotate.Phase.CLEANUP })
public java.lang.String getLocalizedMessage( )
```

Subclasses may override this message to get an error message that is localized to the default locale.

By default it returns getMessage().

returns the error message

```
@SCJMayAllocate({})
@SCJAllowed
@SCJMaySelfSuspend(false)
@Override
@SCJPhase({
    javax.safetycritical.annotate.Phase.STARTUP,
    javax.safetycritical.annotate.Phase.INITIALIZATION,
    javax.safetycritical.annotate.Phase.RUN,
    javax.safetycritical.annotate.Phase.CLEANUP })
public java.lang.String getMessage( )
```

get the message describing the problem from the schedulable object's local memory.

returns the message given to the constructor or null when no message was set.

```
@SCJMayAllocate({})
@SCJAllowed
@SCJMaySelfSuspend(false)
@Override
@SCJPhase({
    javax.safetycritical.annotate.Phase.STARTUP,
    javax.safetycritical.annotate.Phase.INITIALIZATION,
    javax.safetycritical.annotate.Phase.RUN,
    javax.safetycritical.annotate.Phase.CLEANUP })
public java.lang.StackTraceElement[] getStackTrace( )
```

Get the stack trace created by fillInStackTrace for this Throwable as an array of StackTraceElements.

The stack trace does not need to contain entries for all methods that are actually on the call stack, the infrastructure may decide to skip some stack trace entries. Even an empty array is a valid result of this function.

Repeated calls of this function without intervening calls to fillInStackTrace will return the same result.

When memory areas are used, and this Throwable was allocated in a different memory area than the current allocation context, the resulting stack trace will be allocated in either the same memory area this was allocated in or the current memory area, depending on which is the least deeply nested, thereby creating objects that are assignment compatible with both areas.

returns array representing the stack trace, it is never null.

```
@SCJMayAllocate({})
@SCJAllowed
```

```
@SCJMaySelfSuspend(false)
@Override
@SCJPhase({
    javax.safetycritical.annotate.Phase.STARTUP,
    javax.safetycritical.annotate.Phase.INITIALIZATION,
    javax.safetycritical.annotate.Phase.RUN,
    javax.safetycritical.annotate.Phase.CLEANUP })
public java.lang.Throwable initCause(Throwable causingThrowable)
```

Initializes the cause to the given Throwable in the schedulable object's local memory.

causingThrowable — the reason why this Throwable gets Thrown.

returns the reference to this Throwable.

Throws IllegalArgumentException when the cause is this Throwable itself.

```
@SCJMayAllocate({})
@SCJAllowed
@SCJMaySelfSuspend(false)
@Override
@SCJPhase({
    javax.safetycritical.annotate.Phase.STARTUP,
    javax.safetycritical.annotate.Phase.INITIALIZATION,
    javax.safetycritical.annotate.Phase.RUN,
    javax.safetycritical.annotate.Phase.CLEANUP })
public java.lang.Throwable initMessage(String message)
```

Set the message in the schedulable object's local storage. This is the only method that is not also defined in Throwable.

message — is the text to set.

```
@SCJMayAllocate({})
@SCJAllowed
@SCJMaySelfSuspend(false)
@Override
@SCJPhase({
    javax.safetycritical.annotate.Phase.STARTUP,
    javax.safetycritical.annotate.Phase.INITIALIZATION,
    javax.safetycritical.annotate.Phase.RUN,
    javax.safetycritical.annotate.Phase.CLEANUP })
public void printStackTrace(PrintStream stream)
```

Print stack trace of this Throwable to stream.

The printed stack trace contains the result of toString() as the first line followed by one line for each stack trace element that contains the name of the method or constructor, optionally followed by the source file name and source file line number when available.

```
@SCJMayAllocate({})
@SCJAllowed
@SCJMaySelfSuspend(false)
@Override
@SCJPhase({
    javax.safetycritical.annotate.Phase.STARTUP,
    javax.safetycritical.annotate.Phase.INITIALIZATION,
    javax.safetycritical.annotate.Phase.RUN,
    javax.safetycritical.annotate.Phase.CLEANUP })
public void printStackTrace( )
```

Print stack trace of this Throwable to System.err.

The printed stack trace contains the result of toString() as the first line followed by one line for each stack trace element that contains the name of the method or constructor, optionally followed by the source file name and source file line number when available.

```
@SCJMayAllocate({})
@SCJAllowed
@SCJMaySelfSuspend(false)
@Override
@SCJPhase({
    javax.safetycritical.annotate.Phase.STARTUP,
    javax.safetycritical.annotate.Phase.INITIALIZATION,
    javax.safetycritical.annotate.Phase.RUN,
    javax.safetycritical.annotate.Phase.CLEANUP })
public void setStackTrace(StackTraceElement [] new_stackTrace)
  throws java.lang.NullPointerException
```

This method allows overriding the stack trace that was filled during construction of this object. It is intended to be used in a serialization context when the stack trace of a remote exception should be treated like a local.

new_stackTrace — the stack trace to replace be used.

Throws NullPointerException when new_stackTrace or any element of new_stackTrace is null.

D.3.38 CLASS StaticRuntimeException

```
@SCJAllowed
public abstract class StaticRuntimeException implements
    javax.realtime.StaticThrowable extends java.lang.RuntimeException
```

Methods

```
@SCJMayAllocate({})
@SCJAllowed
@SCJMaySelfSuspend(false)
@Override
@SCJPhase({
   javax.safetycritical.annotate.Phase.STARTUP,
   javax.safetycritical.annotate.Phase.INITIALIZATION,
   javax.safetycritical.annotate.Phase.RUN,
   javax.safetycritical.annotate.Phase.CLEANUP })
public java.lang.Throwable fillInStackTrace( )
```

Calls the infrastructure to capture the current stack trace in the schedulable object's local memory.

returns a reference to this Throwable.

```
@SCJMayAllocate({})
@SCJAllowed
@SCJMaySelfSuspend(false)
@SCJPhase({
   javax.safetycritical.annotate.Phase.STARTUP,
   javax.safetycritical.annotate.Phase.INITIALIZATION,
   javax.safetycritical.annotate.Phase.RUN,
   javax.safetycritical.annotate.Phase.CLEANUP })
public static javax.realtime.StaticRuntimeException get( )
```

Get the preallocated version of this Throwable. Allocation is done in memory that acts like **javax.realtime.ImmortalMemory** . The message and cause are cleared and the stack trace is filled out.

returns the preallocated exception

```
@SCJMayAllocate({})
@SCJAllowed
@SCJMaySelfSuspend(false)
@Override
@SCJPhase({
   javax.safetycritical.annotate.Phase.STARTUP,
   javax.safetycritical.annotate.Phase.INITIALIZATION,
   javax.safetycritical.annotate.Phase.RUN,
   javax.safetycritical.annotate.Phase.CLEANUP })
public java.lang.Throwable getCause( )
```

getCause returns the cause of this exception or null when no cause was set by initCause. The cause is another exception that was caught just before this exception was thrown.

returns The cause or null.

```
@SCJMayAllocate({})
@SCJAllowed
@SCJMaySelfSuspend(false)
@Override
@SCJPhase({
    javax.safetycritical.annotate.Phase.STARTUP,
    javax.safetycritical.annotate.Phase.INITIALIZATION,
    javax.safetycritical.annotate.Phase.RUN,
    javax.safetycritical.annotate.Phase.CLEANUP })
public java.lang.String getLocalizedMessage( )
```

Subclasses may override this message to get an error message that is localized to the default locale.

By default it returns getMessage().

returns the error message

```
@SCJMayAllocate({})
@SCJAllowed
@SCJMaySelfSuspend(false)
@Override
@SCJPhase({
    javax.safetycritical.annotate.Phase.STARTUP,
    javax.safetycritical.annotate.Phase.INITIALIZATION,
    javax.safetycritical.annotate.Phase.RUN,
    javax.safetycritical.annotate.Phase.CLEANUP })
public java.lang.String getMessage( )
```

get the message describing the problem from the schedulable object's local memory.

returns the message given to the constructor or null when no message was set.

```
@SCJMayAllocate({})
@SCJAllowed
@SCJMaySelfSuspend(false)
@Override
@SCJPhase({
    javax.safetycritical.annotate.Phase.STARTUP,
    javax.safetycritical.annotate.Phase.INITIALIZATION,
    javax.safetycritical.annotate.Phase.RUN,
    javax.safetycritical.annotate.Phase.CLEANUP })
public java.lang.StackTraceElement[] getStackTrace( )
```

Get the stack trace created by fillInStackTrace for this Throwable as an array of StackTraceElements.

The stack trace does not need to contain entries for all methods that are actually on the call stack, the infrastructure may decide to skip some stack trace entries. Even an empty array is a valid result of this function.

Repeated calls of this function without intervening calls to fillInStackTrace will return the same result.

When memory areas are used, and this Throwable was allocated in a different memory area than the current allocation context, the resulting stack trace will be allocated in either the same memory area this was allocated in or the current memory area, depending on which is the least deeply nested, thereby creating objects that are assignment compatible with both areas.

returns array representing the stack trace, it is never null.

```
@SCJMayAllocate({})
@SCJAllowed
@SCJMaySelfSuspend(false)
@Override
@SCJPhase({
    javax.safetycritical.annotate.Phase.STARTUP,
    javax.safetycritical.annotate.Phase.INITIALIZATION,
    javax.safetycritical.annotate.Phase.RUN,
    javax.safetycritical.annotate.Phase.CLEANUP })
public java.lang.Throwable initCause(Throwable causingThrowable)
```

Initializes the cause to the given Throwable in the schedulable object's local memory.

causingThrowable — the reason why this Throwable gets Thrown.

returns the reference to this Throwable.

Throws IllegalArgumentException when the cause is this Throwable itself.

```
@SCJMayAllocate({})
@SCJAllowed
@SCJMaySelfSuspend(false)
@Override
@SCJPhase({
    javax.safetycritical.annotate.Phase.STARTUP,
    javax.safetycritical.annotate.Phase.INITIALIZATION,
    javax.safetycritical.annotate.Phase.RUN,
    javax.safetycritical.annotate.Phase.CLEANUP })
public java.lang.Throwable initMessage(String message)
```

Set the message in the schedulable object's local storage. This is the only method that is not also defined in Throwable.

message — is the text to set.

```
@SCJMayAllocate({})
@SCJAllowed
@SCJMaySelfSuspend(false)
@Override
@SCJPhase({
    javax.safetycritical.annotate.Phase.STARTUP,
    javax.safetycritical.annotate.Phase.INITIALIZATION,
    javax.safetycritical.annotate.Phase.RUN,
    javax.safetycritical.annotate.Phase.CLEANUP })
public void printStackTrace(PrintStream stream)
```

Print stack trace of this Throwable to stream.

The printed stack trace contains the result of toString() as the first line followed by one line for each stack trace element that contains the name of the method or constructor, optionally followed by the source file name and source file line number when available.

```
@SCJMayAllocate({})
@SCJAllowed
@SCJMaySelfSuspend(false)
@Override
@SCJPhase({
    javax.safetycritical.annotate.Phase.STARTUP,
    javax.safetycritical.annotate.Phase.INITIALIZATION,
    javax.safetycritical.annotate.Phase.RUN,
    javax.safetycritical.annotate.Phase.CLEANUP })
public void printStackTrace( )
```

Print stack trace of this Throwable to System.err.

The printed stack trace contains the result of toString() as the first line followed by one line for each stack trace element that contains the name of the method or constructor, optionally followed by the source file name and source file line number when available.

```
@SCJMayAllocate({})
@SCJAllowed
@SCJMaySelfSuspend(false)
@Override
@SCJPhase({
    javax.safetycritical.annotate.Phase.STARTUP,
    javax.safetycritical.annotate.Phase.INITIALIZATION,
    javax.safetycritical.annotate.Phase.RUN,
```

```
    javax.safetycritical.annotate.Phase.CLEANUP })
public void setStackTrace(StackTraceElement [] new_stackTrace)
  throws java.lang.NullPointerException
```

This method allows overriding the stack trace that was filled during construction of this object. It is intended to be used in a serialization context when the stack trace of a remote exception should be treated like a local.

new_stackTrace — the stack trace to replace be used.

Throws NullPointerException when new_stackTrace or any element of new_stackTrace is null.

D.3.39 CLASS StaticThrowableStorage

```
@SCJAllowed
public class StaticThrowableStorage implements javax.realtime.StaticThrowable
    extends java.lang.Throwable
```

Provide the methods for managing the thread local memory used for storing the data needed by preallocated throwables, i.e., exceptions and errors which implement StaticThrowable. This call is visible so that an application can extend an existing conventional Java throwable and still implement StaticThrowable; its methods can be implemented using the methods defined in this class. An application defined throwable that does not need to extend an existing conventional Java throwable should extend on of StaticCheckedException, StaticRuntimeException, or StaticError instead.

Methods

```
@Override
@SCJMayAllocate({})
@SCJAllowed
@SCJMaySelfSuspend(false)
@SCJPhase({
    javax.safetycritical.annotate.Phase.STARTUP,
    javax.safetycritical.annotate.Phase.INITIALIZATION,
    javax.safetycritical.annotate.Phase.RUN,
    javax.safetycritical.annotate.Phase.CLEANUP })
public java.lang.Throwable fillInStackTrace( )
```

Capture the current thread's stack trace and save it in thread local storage. Only the part of the stack trace that fits in the preallocated buffer is stored. This method should be called by a preallocated exception to implement its method of the same name.

returns this

```
@Override
@SCJMayAllocate({})
@SCJAllowed
@SCJMaySelfSuspend(false)
@SCJPhase({
    javax.safetycritical.annotate.Phase.STARTUP,
    javax.safetycritical.annotate.Phase.INITIALIZATION,
    javax.safetycritical.annotate.Phase.RUN,
    javax.safetycritical.annotate.Phase.CLEANUP })
public java.lang.Throwable getCause( )
```

Get the cause from thread local storage that was saved by the last preallocated exception thrown. The actual exception that of the cause is not saved, but just a reference to its type. This returns a newly allocated exception without any valid content, i.e., no valid stack trace. This method should be called by a preallocated exception to implement its method of the same name.

returns the message

```
@SCJMayAllocate({})
@SCJAllowed
@SCJMaySelfSuspend(false)
@SCJPhase({
    javax.safetycritical.annotate.Phase.STARTUP,
    javax.safetycritical.annotate.Phase.INITIALIZATION,
    javax.safetycritical.annotate.Phase.RUN,
    javax.safetycritical.annotate.Phase.CLEANUP })
public static javax.realtime.StaticThrowableStorage getCurrent( )
```

A means of obtaining the storage object for the current task.

returns the storage object for the current task.

```
@Override
@SCJMayAllocate({})
@SCJAllowed
@SCJMaySelfSuspend(false)
@SCJPhase({
    javax.safetycritical.annotate.Phase.STARTUP,
    javax.safetycritical.annotate.Phase.INITIALIZATION,
    javax.safetycritical.annotate.Phase.RUN,
    javax.safetycritical.annotate.Phase.CLEANUP })
public java.lang.String getLocalizedMessage( )
```

```
@Override
@SCJMayAllocate({})
@SCJAllowed
@SCJMaySelfSuspend(false)
@SCJPhase({
   javax.safetycritical.annotate.Phase.STARTUP,
   javax.safetycritical.annotate.Phase.INITIALIZATION,
   javax.safetycritical.annotate.Phase.RUN,
   javax.safetycritical.annotate.Phase.CLEANUP })
public java.lang.String getMessage( )
```

Get the message from thread local storage that was saved by the last preallocated exception thrown. This method should be called by a preallocated exception to implement its method of the same name.

returns the message

```
@Override
@SCJMayAllocate({})
@SCJAllowed
@SCJMaySelfSuspend(false)
@SCJPhase({
   javax.safetycritical.annotate.Phase.STARTUP,
   javax.safetycritical.annotate.Phase.INITIALIZATION,
   javax.safetycritical.annotate.Phase.RUN,
   javax.safetycritical.annotate.Phase.CLEANUP })
public java.lang.StackTraceElement[] getStackTrace( )
```

Get the stack trace from thread local storage that was saved by the last preallocated exception thrown. This method should be called by a preallocated exception to implement its method of the same name.

returns an array of the elements of the stack trace.

```
@Override
@SCJMayAllocate({})
@SCJAllowed
@SCJMaySelfSuspend(false)
@SCJPhase({
   javax.safetycritical.annotate.Phase.STARTUP,
   javax.safetycritical.annotate.Phase.INITIALIZATION,
   javax.safetycritical.annotate.Phase.RUN,
   javax.safetycritical.annotate.Phase.CLEANUP })
public java.lang.Throwable initCause(Throwable causingThrowable)
```

Save the message in thread local storage for later retrieval. Only a reference to the exception class is stored. The rest of its information is lost. This method should be called by a preallocated exception to implement its method of the same name.

causingThrowable —

returns this

```
@Override
@SCJMayAllocate({})
@SCJAllowed
@SCJMaySelfSuspend(false)
@SCJPhase({
    javax.safetycritical.annotate.Phase.STARTUP,
    javax.safetycritical.annotate.Phase.INITIALIZATION,
    javax.safetycritical.annotate.Phase.RUN,
    javax.safetycritical.annotate.Phase.CLEANUP })
public java.lang.Throwable initMessage(String message)
```

Save the message in thread local storage for later retrieval. Only the part of the message that fits in the preallocated buffer is stored. This method should be called by a preallocated exception to implement its method of the same name.

message — the message to save.

```
@SCJMayAllocate({})
@SCJAllowed
@SCJMaySelfSuspend(false)
@Override
@SCJPhase({
    javax.safetycritical.annotate.Phase.STARTUP,
    javax.safetycritical.annotate.Phase.INITIALIZATION,
    javax.safetycritical.annotate.Phase.RUN,
    javax.safetycritical.annotate.Phase.CLEANUP })
public void printStackTrace(PrintStream stream)
```

```
@Override
@SCJMayAllocate({})
@SCJAllowed
@SCJMaySelfSuspend(false)
@SCJPhase({
    javax.safetycritical.annotate.Phase.STARTUP,
    javax.safetycritical.annotate.Phase.INITIALIZATION,
    javax.safetycritical.annotate.Phase.RUN,
    javax.safetycritical.annotate.Phase.CLEANUP })
public void printStackTrace( )
```

```
@Override
@SCJMayAllocate({})
@SCJAllowed
@SCJMaySelfSuspend(false)
```

```
@SCJPhase({
    javax.safetycritical.annotate.Phase.STARTUP,
    javax.safetycritical.annotate.Phase.INITIALIZATION,
    javax.safetycritical.annotate.Phase.RUN,
    javax.safetycritical.annotate.Phase.CLEANUP })
public void setStackTrace(StackTraceElement [] new_stackTrace)
```

D.3.40 CLASS ThrowBoundaryError

```
@SCJAllowed
public class ThrowBoundaryError implements java.io.Serializable extends
      javax.realtime.StaticError
```

Methods

```
@SCJMayAllocate({javax.safetycritical.annotate.AllocationContext.CURRENT})
@SCJAllowed
@SCJMaySelfSuspend(false)
@SCJPhase({
    javax.safetycritical.annotate.Phase.STARTUP,
    javax.safetycritical.annotate.Phase.INITIALIZATION,
    javax.safetycritical.annotate.Phase.RUN,
    javax.safetycritical.annotate.Phase.CLEANUP })
public static javax.realtime.ThrowBoundaryError get( )
```

Get the preallocated instance of this exception.

returns the preallocated instance of this exception.

Appendix E

Javadoc Description of Package javax.realtime.device

Package Contents *Page*

Classes

E.1 Classes

E.2 Interfaces

E.2.1 INTERFACE **RawByte**

@SCJAllowed
public interface RawByte **extends** javax.realtime.device.RawByteReader, javax.realtime.device.RawByteWriter

A marker for an object that can be used to access to a single byte. Read and write access to that byte is checked by the factory that creates the instance; therefore, no access checking is provided by this interface, only bounds checking.

Since
RTSJ 2.0

E.2.2 INTERFACE **RawByteReader**

@SCJAllowed
public interface RawByteReader **extends** javax.realtime.device.RawMemory

A marker for a byte accessor object encapsulating the protocol for reading bytes from raw memory. A byte accessor can always access at least one byte. Each byte is transfered in a single atomic operation. Groups of bytes may be transfered together; however, this is not required.

Objects of this type are created with the method **javax.realtime.device.RawMemoryFactorycreateRawByteReader** and **javax.realtime.device.RawMemoryFactorycreateRawByte** . Each object references a range of elements in the **javax.realtime.device.RawMemoryRegion** starting at the *base address* provided to the factory method. The size provided to the factor method determines the number of elements accessable.

Caching of the memory access is controlled by the factory that created this object. If the memory is not cached, this method guarantees serialized access. In other words, the memory access at the memory occurs in the same order as in the program. Multiple writes to the same location may not be coalesced.

Since
RTSJ 2.0

Methods

```
@SCJAllowed
@SCJMayAllocate({})
@SCJMaySelfSuspend(false)
@SCJPhase({
    javax.safetycritical.annotate.Phase.STARTUP,
    javax.safetycritical.annotate.Phase.INITIALIZATION,
    javax.safetycritical.annotate.Phase.RUN,
    javax.safetycritical.annotate.Phase.CLEANUP })
public int get(int offset, byte [] values)
  throws javax.realtime.OffsetOutOfBoundsException,
      java.lang.NullPointerException
```

Fill values with elements from this instance, where the nth element is at the address: base address + (offset+n) x stride x element size in bytes. Only the bytes in the intersection of the start and end of values and the *base address* and the end of the memory region are transfered. When an exception is thrown, no data is transfered.

offset — of the first byte in the memory region to transfere

values — the array to receive the bytes

returns the number of elements actuall transferred to values

Throws OffsetOutOfBoundsException when offset is negative or greater than or equal to the number of elements in the raw memory region.

Throws NullPointerException when values is null.

```
@SCJAllowed
@SCJMayAllocate({})
@SCJMaySelfSuspend(false)
@SCJPhase({
    javax.safetycritical.annotate.Phase.STARTUP,
    javax.safetycritical.annotate.Phase.INITIALIZATION,
    javax.safetycritical.annotate.Phase.RUN,
    javax.safetycritical.annotate.Phase.CLEANUP })
public int get(int offset, byte [] values, int start, int count)
  throws javax.realtime.OffsetOutOfBoundsException,
      java.lang.ArrayIndexOutOfBoundsException, java.lang.NullPointerException
```

Fill values from index start with elements from this instance, where the nth element is at the address: base address + (offset+n) x stride x element size in bytes. The number of bytes transfered is the minimum of count, the *size* of the memory region minus offset, and length of values minus start. When an exception is thrown, no data is transfered.

offset — of the first byte in the memory region to transfere

values — the array to receive the bytes

start — the first index in array to fill

count — the maximum number of bytes to copy

returns the number of bytes actually transfered.

Throws OffsetOutOfBoundsException when offset is negative or either offset or offset + count is greater than or equal to the size of this raw memory area.

Throws ArrayIndexOutOfBoundsException when start is negative or either start or start + count is greater than or equal to the size of values.

Throws NullPointerException when values is null or count is negative.

```
@SCJAllowed
@SCJMayAllocate({})
@SCJMaySelfSuspend(false)
@SCJPhase({
    javax.safetycritical.annotate.Phase.STARTUP,
    javax.safetycritical.annotate.Phase.INITIALIZATION,
    javax.safetycritical.annotate.Phase.RUN,
    javax.safetycritical.annotate.Phase.CLEANUP })
public byte getByte(int offset)
  throws javax.realtime.OffsetOutOfBoundsException
```

Get the value at the address: base address + offset x stride x element size in bytes. When an exception is thrown, no data is transfered.

offset — of byte in the memory region starting from the address specified in the associated factory method.

returns the value at the address specified.

Throws OffsetOutOfBoundsException when offset is negative or greater than or equal to the number of elements in the raw memory region.

```
@SCJAllowed
@SCJMayAllocate({})
@SCJMaySelfSuspend(false)
@SCJPhase({
    javax.safetycritical.annotate.Phase.STARTUP,
    javax.safetycritical.annotate.Phase.INITIALIZATION,
    javax.safetycritical.annotate.Phase.RUN,
    javax.safetycritical.annotate.Phase.CLEANUP })
public byte getByte( )
```

Get the value at the first position referenced by this instance, i.e., the value at its start address. This operation must be atomic with respect to all other raw memory accesses to the address.

returns the value at the *base address*.

E.2.3 INTERFACE **RawByteWriter**

```
@SCJAllowed
public interface RawByteWriter extends javax.realtime.device.RawMemory
```

A marker for a byte accessor object encapsulating the protocol for writing bytes to raw memory. A byte accessor can always access at least one byte. Each byte is transfered in a single atomic operation. Groups of bytes may be transfered together; however, this is not required.

Objects of this type are created with the method **javax.realtime.device.RawMemoryFactorycreateRawByteWriter** and **javax.realtime.device.RawMemoryFactorycreateRawByte** . Each object references a range of elements in the **javax.realtime.device.RawMemoryRegion** starting at the *base address* provided to the factory method. The size provided to the factor method determines the number of elements accessable.

Caching of the memory access is controlled by the factory that created this object. If the memory is not cached, this method guarantees serialized access. In other words, the memory access at the memory occurs in the same order as in the program. Multiple writes to the same location may not be coalesced.

Since
RTSJ 2.0

Methods

```
@SCJAllowed
@SCJMayAllocate({})
@SCJMaySelfSuspend(false)
@SCJPhase({
    javax.safetycritical.annotate.Phase.STARTUP,
    javax.safetycritical.annotate.Phase.INITIALIZATION,
    javax.safetycritical.annotate.Phase.RUN,
    javax.safetycritical.annotate.Phase.CLEANUP })
public int set(int offset, byte [] values)
  throws javax.realtime.OffsetOutOfBoundsException,
      java.lang.NullPointerException
```

Copy from values to the memory region from index start,to elements where the nth element is at the address: base address + (offset+n) x stride x element size in bytes. Only the bytes in the intersection of values and the end of the memory region are transfered. When an exception is thrown, no data is transfered.

offset — of first byte in the memory region to be set.

values — is the source of the data to write.

returns the number of elements actually transferred to values

Throws OffsetOutOfBoundsException when offset is negative or greater than or equal to the number of elements in the raw memory region.

Throws NullPointerException when values is null.

```
@SCJAllowed
@SCJMayAllocate({})
@SCJMaySelfSuspend(false)
@SCJPhase({
    javax.safetycritical.annotate.Phase.STARTUP,
    javax.safetycritical.annotate.Phase.INITIALIZATION,
    javax.safetycritical.annotate.Phase.RUN,
    javax.safetycritical.annotate.Phase.CLEANUP })
public int set(int offset, byte [] values, int start, int count)
  throws javax.realtime.OffsetOutOfBoundsException,
     java.lang.ArrayIndexOutOfBoundsException, java.lang.NullPointerException
```

Copy values to the memory region, where offset is first byte in the memory region to write and start is the first index in values from which to read. The number of bytes transfered is the minimum of count, the *size* of the memory region minus offset, and length of values minus start. When an exception is thrown, no data is transfered.

offset — of the first byte in the memory region to set

values — the array from which to retrieve the bytes

start — the first index in array to copy

count — the maximum number of bytes to copy

returns the number of bytes actually transfered.

Throws OffsetOutOfBoundsException when offset is negative or either offset or offset + count is greater than or equal to the size of this raw memory area.

Throws ArrayIndexOutOfBoundsException when start is negative or either start or start + count is greater than or equal to the size of values.

Throws NullPointerException when values is null.

```
@SCJAllowed
@SCJMayAllocate({})
@SCJMaySelfSuspend(false)
@SCJPhase({
    javax.safetycritical.annotate.Phase.STARTUP,
    javax.safetycritical.annotate.Phase.INITIALIZATION,
    javax.safetycritical.annotate.Phase.RUN,
    javax.safetycritical.annotate.Phase.CLEANUP })
public void setByte(int offset, byte value)
  throws javax.realtime.OffsetOutOfBoundsException
```

Set the value of the n^{th} element referenced by this instance, where n is offset and the address is *base address* + offset x *size of Byte*. This operation must be atomic with respect to all other raw memory accesses to the address. When an exception is thrown, no data is transfered.

offset — of byte in the memory region.

value — is the new value for the element.

Throws OffsetOutOfBoundsException when offset is negative or greater than or equal to the number of elements in the raw memory region.

```
@SCJAllowed
@SCJMayAllocate({})
@SCJMaySelfSuspend(false)
@SCJPhase({
    javax.safetycritical.annotate.Phase.STARTUP,
    javax.safetycritical.annotate.Phase.INITIALIZATION,
    javax.safetycritical.annotate.Phase.RUN,
    javax.safetycritical.annotate.Phase.CLEANUP })
public void setByte(byte value)
```

Set the value at the first position referenced by this instance, i.e., the value at its start address. This operation must be atomic with respect to all other raw memory accesses to the address.

value — is the new value for the element.

E.2.4 INTERFACE **RawDouble**

```
@SCJAllowed
public interface RawDouble extends javax.realtime.device.RawDoubleReader,
    javax.realtime.device.RawDoubleWriter
```

A marker for an object that can be used to access to a single double. Read and write access to that double is checked by the factory that creates the instance; therefore, no access checking is provided by this interface, only bounds checking.

Since
RTSJ 2.0

E.2.5 INTERFACE **RawDoubleReader**

@SCJAllowed
public interface RawDoubleReader **extends** javax.realtime.device.RawMemory

A marker for a double accessor object encapsulating the protocol for reading doubles from raw memory. A double accessor can always access at least one double. Each double is transfered in a single atomic operation. Groups of doubles may be transfered together; however, this is not required.

Objects of this type are created with the method **javax.realtime.device.RawMemoryFactorycreateRawDoubleReader** and **javax.realtime.device.RawMemoryFactorycreateRawDouble** . Each object references a range of elements in the **javax.realtime.device.RawMemoryRegion** starting at the *base address* provided to the factory method. The size provided to the factor method determines the number of elements accessable.

Caching of the memory access is controlled by the factory that created this object. If the memory is not cached, this method guarantees serialized access. In other words, the memory access at the memory occurs in the same order as in the program. Multiple writes to the same location may not be coalesced.

Since
RTSJ 2.0

Methods

```
@SCJAllowed
@SCJMayAllocate({})
@SCJMaySelfSuspend(false)
@SCJPhase({
   javax.safetycritical.annotate.Phase.STARTUP,
   javax.safetycritical.annotate.Phase.INITIALIZATION,
   javax.safetycritical.annotate.Phase.RUN,
   javax.safetycritical.annotate.Phase.CLEANUP })
```

```
public int get(int offset, double [] values)
  throws javax.realtime.OffsetOutOfBoundsException,
     java.lang.NullPointerException
```

Fill values with elements from this instance, where the nth element is at the address: base address + (offset+n) x stride x element size in bytes. Only the doubles in the intersection of the start and end of values and the *base address* and the end of the memory region are transfered. When an exception is thrown, no data is transfered.

offset — of the first double in the memory region to transfere

values — the array to receive the doubles

returns the number of elements actuall transferred to values

Throws OffsetOutOfBoundsException when offset is negative or greater than or equal to the number of elements in the raw memory region.

Throws NullPointerException when values is null.

```
@SCJAllowed
@SCJMayAllocate({})
@SCJMaySelfSuspend(false)
@SCJPhase({
    javax.safetycritical.annotate.Phase.STARTUP,
    javax.safetycritical.annotate.Phase.INITIALIZATION,
    javax.safetycritical.annotate.Phase.RUN,
    javax.safetycritical.annotate.Phase.CLEANUP })
public int get(int offset, double [] values, int start, int count)
  throws javax.realtime.OffsetOutOfBoundsException,
     java.lang.ArrayIndexOutOfBoundsException, java.lang.NullPointerException
```

Fill values from index start with elements from this instance, where the nth element is at the address: base address + (offset+n) x stride x element size in bytes. The number of bytes transfered is the minimum of count, the *size* of the memory region minus offset, and length of values minus start. When an exception is thrown, no data is transfered.

offset — of the first double in the memory region to transfere

values — the array to receive the doubles

start — the first index in array to fill

count — the maximum number of doubles to copy

returns the number of doubles actually transfered.

Throws OffsetOutOfBoundsException when offset is negative or either offset or offset + count is greater than or equal to the size of this raw memory area.

Throws ArrayIndexOutOfBoundsException when start is negative or either start or start + count is greater than or equal to the size of values.

Throws NullPointerException when values is null or count is negative.

```
@SCJAllowed
@SCJMayAllocate({})
@SCJMaySelfSuspend(false)
@SCJPhase({
    javax.safetycritical.annotate.Phase.STARTUP,
    javax.safetycritical.annotate.Phase.INITIALIZATION,
    javax.safetycritical.annotate.Phase.RUN,
    javax.safetycritical.annotate.Phase.CLEANUP })
public double getDouble(int offset)
  throws javax.realtime.OffsetOutOfBoundsException
```

Get the value at the address: base address + offset x stride x element size in bytes. When an exception is thrown, no data is transfered.

offset — of double in the memory region starting from the address specified in the associated factory method.

returns the value at the address specified.

Throws OffsetOutOfBoundsException when offset is negative or greater than or equal to the number of elements in the raw memory region.

```
@SCJAllowed
@SCJMayAllocate({})
@SCJMaySelfSuspend(false)
@SCJPhase({
    javax.safetycritical.annotate.Phase.STARTUP,
    javax.safetycritical.annotate.Phase.INITIALIZATION,
    javax.safetycritical.annotate.Phase.RUN,
    javax.safetycritical.annotate.Phase.CLEANUP })
public double getDouble( )
```

Get the value at the first position referenced by this instance, i.e., the value at its start address. This operation must be atomic with respect to all other raw memory accesses to the address.

returns the value at the *base address*.

E.2.6 INTERFACE **RawDoubleWriter**

```
@SCJAllowed
public interface RawDoubleWriter extends javax.realtime.device.RawMemory
```

A marker for a double accessor object encapsulating the protocol for writing doubles to raw memory. A double accessor can always access at least one double. Each double is transfered in a single atomic operation. Groups of doubles may be transfered together; however, this is not required.

Objects of this type are created with the method **javax.realtime.device.RawMemoryFactorycreateRawDoubleWriter** and **javax.realtime.device.RawMemoryFactorycreateRawDouble** . Each object references a range of elements in the **javax.realtime.device.RawMemoryRegion** starting at the *base address* provided to the factory method. The size provided to the factor method determines the number of elements accessable.

Caching of the memory access is controlled by the factory that created this object. If the memory is not cached, this method guarantees serialized access. In other words, the memory access at the memory occurs in the same order as in the program. Multiple writes to the same location may not be coalesced.

Since
RTSJ 2.0

Methods

```
@SCJAllowed
@SCJMayAllocate({})
@SCJMaySelfSuspend(false)
@SCJPhase({
    javax.safetycritical.annotate.Phase.STARTUP,
    javax.safetycritical.annotate.Phase.INITIALIZATION,
    javax.safetycritical.annotate.Phase.RUN,
    javax.safetycritical.annotate.Phase.CLEANUP })
public int set(int offset, double [] values)
  throws javax.realtime.OffsetOutOfBoundsException,
      java.lang.NullPointerException
```

Copy from values to the memory region from index start,to elements where the nth element is at the address: base address + (offset+n) x stride x element size in bytes. Only the doubles in the intersection of values and the end of the memory region are transfered. When an exception is thrown, no data is transfered.

offset — of first double in the memory region to be set.

values — is the source of the data to write.

returns the number of elements actually transferred to values

Throws OffsetOutOfBoundsException when offset is negative or greater than or equal to the number of elements in the raw memory region.

Throws NullPointerException when values is null.

```
@SCJAllowed
@SCJMayAllocate({})
@SCJMaySelfSuspend(false)
@SCJPhase({
    javax.safetycritical.annotate.Phase.STARTUP,
    javax.safetycritical.annotate.Phase.INITIALIZATION,
    javax.safetycritical.annotate.Phase.RUN,
    javax.safetycritical.annotate.Phase.CLEANUP })
public int set(int offset, double [] values, int start, int count)
  throws javax.realtime.OffsetOutOfBoundsException,
      java.lang.ArrayIndexOutOfBoundsException, java.lang.NullPointerException
```

Copy values to the memory region, where offset is first double in the memory region to write and start is the first index in values from which to read. The number of bytes transfered is the minimum of count, the *size* of the memory region minus offset, and length of values minus start. When an exception is thrown, no data is transfered.

offset — of the first double in the memory region to set

values — the array from which to retrieve the doubles

start — the first index in array to copy

count — the maximum number of doubles to copy

returns the number of doubles actually transfered.

Throws OffsetOutOfBoundsException when offset is negative or either offset or offset + count is greater than or equal to the size of this raw memory area.

Throws ArrayIndexOutOfBoundsException when start is negative or either start or start + count is greater than or equal to the size of values.

Throws NullPointerException when values is null.

```
@SCJAllowed
@SCJMayAllocate({})
@SCJMaySelfSuspend(false)
@SCJPhase({
    javax.safetycritical.annotate.Phase.STARTUP,
    javax.safetycritical.annotate.Phase.INITIALIZATION,
    javax.safetycritical.annotate.Phase.RUN,
    javax.safetycritical.annotate.Phase.CLEANUP })
public void setDouble(int offset, double value)
  throws javax.realtime.OffsetOutOfBoundsException
```

Set the value of the n^{th} element referenced by this instance, where n is offset and the address is *base address* + offset x *size of Double*. This operation must be atomic with respect to all other raw memory accesses to the address. When an exception is thrown, no data is transfered.

offset — of double in the memory region.

value — is the new value for the element.

Throws OffsetOutOfBoundsException when offset is negative or greater than or equal to the number of elements in the raw memory region.

```
@SCJAllowed
@SCJMayAllocate({})
@SCJMaySelfSuspend(false)
@SCJPhase({
    javax.safetycritical.annotate.Phase.STARTUP,
    javax.safetycritical.annotate.Phase.INITIALIZATION,
    javax.safetycritical.annotate.Phase.RUN,
    javax.safetycritical.annotate.Phase.CLEANUP })
public void setDouble(double value)
```

Set the value at the first position referenced by this instance, i.e., the value at its start address. This operation must be atomic with respect to all other raw memory accesses to the address.

value — is the new value for the element.

E.2.7 INTERFACE **RawFloat**

```
@SCJAllowed
public interface RawFloat extends javax.realtime.device.RawFloatReader,
    javax.realtime.device.RawFloatWriter
```

A marker for an object that can be used to access to a single float. Read and write access to that float is checked by the factory that creates the instance; therefore, no access checking is provided by this interface, only bounds checking.

Since
RTSJ 2.0

E.2.8 INTERFACE **RawFloatReader**

@SCJAllowed
public interface RawFloatReader **extends** javax.realtime.device.RawMemory

A marker for a float accessor object encapsulating the protocol for reading floats from raw memory. A float accessor can always access at least one float. Each float is transfered in a single atomic operation. Groups of floats may be transfered together; however, this is not required.

Objects of this type are created with the method **javax.realtime.device.RawMemoryFactorycreateRawFloatReader** and **javax.realtime.device.RawMemoryFactorycreateRawFloat** . Each object references a range of elements in the **javax.realtime.device.RawMemoryRegion** starting at the *base address* provided to the factory method. The size provided to the factor method determines the number of elements accessable.

Caching of the memory access is controlled by the factory that created this object. If the memory is not cached, this method guarantees serialized access. In other words, the memory access at the memory occurs in the same order as in the program. Multiple writes to the same location may not be coalesced.

Since
RTSJ 2.0

Methods

```
@SCJAllowed
@SCJMayAllocate({})
@SCJMaySelfSuspend(false)
@SCJPhase({
    javax.safetycritical.annotate.Phase.STARTUP,
    javax.safetycritical.annotate.Phase.INITIALIZATION,
    javax.safetycritical.annotate.Phase.RUN,
    javax.safetycritical.annotate.Phase.CLEANUP })
public int get(int offset, float [] values)
  throws javax.realtime.OffsetOutOfBoundsException,
      java.lang.NullPointerException
```

Fill values with elements from this instance, where the nth element is at the address: base address + (offset+n) x stride x element size in bytes. Only the floats in the intersection of the start and end of values and the *base address* and the end of the memory region are transfered. When an exception is thrown, no data is transfered.

offset — of the first float in the memory region to transfere

values — the array to receive the floats

returns the number of elements actuall transferred to values

Throws OffsetOutOfBoundsException when offset is negative or greater than or equal to the number of elements in the raw memory region.

Throws NullPointerException when values is null.

```
@SCJAllowed
@SCJMayAllocate({})
@SCJMaySelfSuspend(false)
@SCJPhase({
    javax.safetycritical.annotate.Phase.STARTUP,
    javax.safetycritical.annotate.Phase.INITIALIZATION,
    javax.safetycritical.annotate.Phase.RUN,
    javax.safetycritical.annotate.Phase.CLEANUP })
public int get(int offset, float [] values, int start, int count)
  throws javax.realtime.OffsetOutOfBoundsException,
     java.lang.ArrayIndexOutOfBoundsException, java.lang.NullPointerException
```

Fill values from index start with elements from this instance, where the nth element is at the address: base address + (offset+n) x stride x element size in bytes. The number of bytes transfered is the minimum of count, the *size* of the memory region minus offset, and length of values minus start. When an exception is thrown, no data is transfered.

offset — of the first float in the memory region to transfere

values — the array to receive the floats

start — the first index in array to fill

count — the maximum number of floats to copy

returns the number of floats actually transfered.

Throws OffsetOutOfBoundsException when offset is negative or either offset or offset + count is greater than or equal to the size of this raw memory area.

Throws ArrayIndexOutOfBoundsException when start is negative or either start or start + count is greater than or equal to the size of values.

Throws NullPointerException when values is null or count is negative.

```
@SCJAllowed
@SCJMayAllocate({})
@SCJMaySelfSuspend(false)
@SCJPhase({
    javax.safetycritical.annotate.Phase.STARTUP,
    javax.safetycritical.annotate.Phase.INITIALIZATION,
```

```
    javax.safetycritical.annotate.Phase.RUN,
    javax.safetycritical.annotate.Phase.CLEANUP })
public float getFloat(int offset)
  throws javax.realtime.OffsetOutOfBoundsException
```

Get the value at the address: base address + offset x stride x element size in bytes. When an exception is thrown, no data is transfered.

offset — of float in the memory region starting from the address specified in the associated factory method.

returns the value at the address specified.

Throws OffsetOutOfBoundsException when offset is negative or greater than or equal to the number of elements in the raw memory region.

```
@SCJAllowed
@SCJMayAllocate({})
@SCJMaySelfSuspend(false)
@SCJPhase({
    javax.safetycritical.annotate.Phase.STARTUP,
    javax.safetycritical.annotate.Phase.INITIALIZATION,
    javax.safetycritical.annotate.Phase.RUN,
    javax.safetycritical.annotate.Phase.CLEANUP })
public float getFloat( )
```

Get the value at the first position referenced by this instance, i.e., the value at its start address. This operation must be atomic with respect to all other raw memory accesses to the address.

returns the value at the *base address*.

E.2.9 INTERFACE **RawFloatWriter**

```
@SCJAllowed
public interface RawFloatWriter extends javax.realtime.device.RawMemory
```

A marker for a float accessor object encapsulating the protocol for writing floats to raw memory. A float accessor can always access at least one float. Each float is transfered in a single atomic operation. Groups of floats may be transfered together; however, this is not required.

Objects of this type are created with the method **javax.realtime.device.RawMemoryFactorycreateRawFloatWriter** and **javax.realtime.device.RawMemoryFactorycreateRawFloat** . Each object references a range of elements in the

javax.realtime.device.RawMemoryRegion starting at the *base address* provided to the factory method. The size provided to the factor method determines the number of elements accessable.

Caching of the memory access is controlled by the factory that created this object. If the memory is not cached, this method guarantees serialized access. In other words, the memory access at the memory occurs in the same order as in the program. Multiple writes to the same location may not be coalesced.

Since
RTSJ 2.0

Methods

```
@SCJAllowed
@SCJMayAllocate({})
@SCJMaySelfSuspend(false)
@SCJPhase({
    javax.safetycritical.annotate.Phase.STARTUP,
    javax.safetycritical.annotate.Phase.INITIALIZATION,
    javax.safetycritical.annotate.Phase.RUN,
    javax.safetycritical.annotate.Phase.CLEANUP })
public int set(int offset, float [] values)
  throws javax.realtime.OffsetOutOfBoundsException,
      java.lang.NullPointerException
```

Copy from values to the memory region from index start,to elements where the nth element is at the address: base address + (offset+n) x stride x element size in bytes. Only the floats in the intersection of values and the end of the memory region are transfered. When an exception is thrown, no data is transfered.

offset — of first float in the memory region to be set.

values — is the source of the data to write.

returns the number of elements actually transferred to values

Throws OffsetOutOfBoundsException when offset is negative or greater than or equal to the number of elements in the raw memory region.

Throws NullPointerException when values is null.

```
@SCJAllowed
@SCJMayAllocate({})
@SCJMaySelfSuspend(false)
@SCJPhase({
    javax.safetycritical.annotate.Phase.STARTUP,
```

```
    javax.safetycritical.annotate.Phase.INITIALIZATION,
    javax.safetycritical.annotate.Phase.RUN,
    javax.safetycritical.annotate.Phase.CLEANUP })
public int set(int offset, float [] values, int start, int count)
  throws javax.realtime.OffsetOutOfBoundsException,
      java.lang.ArrayIndexOutOfBoundsException, java.lang.NullPointerException
```

Copy values to the memory region, where offset is first float in the memory region to write and start is the first index in values from which to read. The number of bytes transfered is the minimum of count, the *size* of the memory region minus offset, and length of values minus start. When an exception is thrown, no data is transfered.

offset — of the first float in the memory region to set

values — the array from which to retrieve the floats

start — the first index in array to copy

count — the maximum number of floats to copy

returns the number of floats actually transfered.

Throws OffsetOutOfBoundsException when offset is negative or either offset or offset + count is greater than or equal to the size of this raw memory area.

Throws ArrayIndexOutOfBoundsException when start is negative or either start or start + count is greater than or equal to the size of values.

Throws NullPointerException when values is null.

```
@SCJAllowed
@SCJMayAllocate({})
@SCJMaySelfSuspend(false)
@SCJPhase({
    javax.safetycritical.annotate.Phase.STARTUP,
    javax.safetycritical.annotate.Phase.INITIALIZATION,
    javax.safetycritical.annotate.Phase.RUN,
    javax.safetycritical.annotate.Phase.CLEANUP })
public void setFloat(int offset, float value)
  throws javax.realtime.OffsetOutOfBoundsException
```

Set the value of the n^{th} element referenced by this instance, where n is offset and the address is *base address* + offset x *size of Float*. This operation must be atomic with respect to all other raw memory accesses to the address. When an exception is thrown, no data is transfered.

offset — of float in the memory region.

value — is the new value for the element.

Throws OffsetOutOfBoundsException when offset is negative or greater than or equal to the number of elements in the raw memory region.

```
@SCJAllowed
@SCJMayAllocate({})
@SCJMaySelfSuspend(false)
@SCJPhase({
    javax.safetycritical.annotate.Phase.STARTUP,
    javax.safetycritical.annotate.Phase.INITIALIZATION,
    javax.safetycritical.annotate.Phase.RUN,
    javax.safetycritical.annotate.Phase.CLEANUP })
public void setFloat(float value)
```

Set the value at the first position referenced by this instance, i.e., the value at its start address. This operation must be atomic with respect to all other raw memory accesses to the address.

value — is the new value for the element.

E.2.10 INTERFACE **RawInt**

```
@SCJAllowed
public interface RawInt extends javax.realtime.device.RawIntReader,
    javax.realtime.device.RawIntWriter
```

A marker for an object that can be used to access to a single int. Read and write access to that int is checked by the factory that creates the instance; therefore, no access checking is provided by this interface, only bounds checking.

Since
RTSJ 2.0

E.2.11 INTERFACE **RawIntReader**

```
@SCJAllowed
public interface RawIntReader extends javax.realtime.device.RawMemory
```

A marker for a int accessor object encapsulating the protocol for reading ints from raw memory. A int accessor can always access at least one int. Each int is transfered in a single atomic operation. Groups of ints may be transfered together; however, this is not required.

Objects of this type are created with the method **javax.realtime.device.RawMemoryFactorycreateRawIntReader** and **javax.realtime.device.RawMemoryFactorycreateRawInt** . Each object references a range of elements in the **javax.realtime.device.RawMemoryRegion** starting at the *base address* provided to the factory method. The size provided to the factor method determines the number of elements accessable.

Caching of the memory access is controlled by the factory that created this object. If the memory is not cached, this method guarantees serialized access. In other words, the memory access at the memory occurs in the same order as in the program. Multiple writes to the same location may not be coalesced.

Since
RTSJ 2.0

Methods

```
@SCJAllowed
@SCJMayAllocate({})
@SCJMaySelfSuspend(false)
@SCJPhase({
    javax.safetycritical.annotate.Phase.STARTUP,
    javax.safetycritical.annotate.Phase.INITIALIZATION,
    javax.safetycritical.annotate.Phase.RUN,
    javax.safetycritical.annotate.Phase.CLEANUP })
public int get(int offset, int [] values)
  throws javax.realtime.OffsetOutOfBoundsException,
      java.lang.NullPointerException
```

Fill values with elements from this instance, where the nth element is at the address: base address + (offset+n) x stride x element size in bytes. Only the ints in the intersection of the start and end of values and the *base address* and the end of the memory region are transfered. When an exception is thrown, no data is transfered.

offset — of the first int in the memory region to transfere

values — the array to receive the ints

returns the number of elements actuall transferred to values

Throws OffsetOutOfBoundsException when offset is negative or greater than or equal to the number of elements in the raw memory region.

Throws NullPointerException when values is null.

```
@SCJAllowed
@SCJMayAllocate({})
@SCJMaySelfSuspend(false)
@SCJPhase({
    javax.safetycritical.annotate.Phase.STARTUP,
    javax.safetycritical.annotate.Phase.INITIALIZATION,
    javax.safetycritical.annotate.Phase.RUN,
    javax.safetycritical.annotate.Phase.CLEANUP })
public int get(int offset, int [] values, int start, int count)
  throws javax.realtime.OffsetOutOfBoundsException,
     java.lang.ArrayIndexOutOfBoundsException, java.lang.NullPointerException
```

Fill values from index start with elements from this instance, where the nth element is at the address: base address + (offset+n) x stride x element size in bytes. The number of bytes transfered is the minimum of count, the *size* of the memory region minus offset, and length of values minus start. When an exception is thrown, no data is transfered.

offset — of the first int in the memory region to transfere

values — the array to receive the ints

start — the first index in array to fill

count — the maximum number of ints to copy

returns the number of ints actually transfered.

Throws OffsetOutOfBoundsException when offset is negative or either offset or offset + count is greater than or equal to the size of this raw memory area.

Throws ArrayIndexOutOfBoundsException when start is negative or either start or start + count is greater than or equal to the size of values.

Throws NullPointerException when values is null or count is negative.

```
@SCJAllowed
@SCJMayAllocate({})
@SCJMaySelfSuspend(false)
@SCJPhase({
    javax.safetycritical.annotate.Phase.STARTUP,
    javax.safetycritical.annotate.Phase.INITIALIZATION,
    javax.safetycritical.annotate.Phase.RUN,
    javax.safetycritical.annotate.Phase.CLEANUP })
public int getInt(int offset)
  throws javax.realtime.OffsetOutOfBoundsException
```

Get the value at the address: base address + offset x stride x element size in bytes. When an exception is thrown, no data is transfered.

offset — of int in the memory region starting from the address specified in the associated factory method.

returns the value at the address specified.

Throws OffsetOutOfBoundsException when offset is negative or greater than or equal to the number of elements in the raw memory region.

```
@SCJAllowed
@SCJMayAllocate({})
@SCJMaySelfSuspend(false)
@SCJPhase({
   javax.safetycritical.annotate.Phase.STARTUP,
   javax.safetycritical.annotate.Phase.INITIALIZATION,
   javax.safetycritical.annotate.Phase.RUN,
   javax.safetycritical.annotate.Phase.CLEANUP })
public int getInt( )
```

Get the value at the first position referenced by this instance, i.e., the value at its start address. This operation must be atomic with respect to all other raw memory accesses to the address.

returns the value at the *base address*.

E.2.12 INTERFACE **RawIntWriter**

```
@SCJAllowed
public interface RawIntWriter extends javax.realtime.device.RawMemory
```

A marker for a int accessor object encapsulating the protocol for writing ints to raw memory. A int accessor can always access at least one int. Each int is transfered in a single atomic operation. Groups of ints may be transfered together; however, this is not required.

Objects of this type are created with the method **javax.realtime.device.RawMemoryFactorycreateRawIntWriter** and **javax.realtime.device.RawMemoryFactorycreateRawInt** . Each object references a range of elements in the **javax.realtime.device.RawMemoryRegion** starting at the *base address* provided to the factory method. The size provided to the factor method determines the number of elements accessable.

Caching of the memory access is controlled by the factory that created this object. If the memory is not cached, this method guarantees serialized access. In other words, the memory access at the memory occurs in the same order as in the program. Multiple writes to the same location may not be coalesced.

Since
RTSJ 2.0

Methods

```
@SCJAllowed
@SCJMayAllocate({})
@SCJMaySelfSuspend(false)
@SCJPhase({
    javax.safetycritical.annotate.Phase.STARTUP,
    javax.safetycritical.annotate.Phase.INITIALIZATION,
    javax.safetycritical.annotate.Phase.RUN,
    javax.safetycritical.annotate.Phase.CLEANUP })
public int set(int offset, int [] values)
  throws javax.realtime.OffsetOutOfBoundsException,
     java.lang.NullPointerException
```

Copy from values to the memory region from index start,to elements where the nth element is at the address: base address + (offset+n) x stride x element size in bytes. Only the ints in the intersection of values and the end of the memory region are transfered. When an exception is thrown, no data is transfered.

offset — of first int in the memory region to be set.

values — is the source of the data to write.

returns the number of elements actually transferred to values

Throws OffsetOutOfBoundsException when offset is negative or greater than or equal to the number of elements in the raw memory region.

Throws NullPointerException when values is null.

```
@SCJAllowed
@SCJMayAllocate({})
@SCJMaySelfSuspend(false)
@SCJPhase({
    javax.safetycritical.annotate.Phase.STARTUP,
    javax.safetycritical.annotate.Phase.INITIALIZATION,
    javax.safetycritical.annotate.Phase.RUN,
    javax.safetycritical.annotate.Phase.CLEANUP })
public int set(int offset, int [] values, int start, int count)
  throws javax.realtime.OffsetOutOfBoundsException,
     java.lang.ArrayIndexOutOfBoundsException, java.lang.NullPointerException
```

Copy values to the memory region, where offset is first int in the memory region to write and start is the first index in values from which to read. The

number of bytes transfered is the minimum of count, the *size* of the memory region minus offset, and length of values minus start. When an exception is thrown, no data is transfered.

offset — of the first int in the memory region to set

values — the array from which to retrieve the ints

start — the first index in array to copy

count — the maximum number of ints to copy

returns the number of ints actually transfered.

Throws OffsetOutOfBoundsException when offset is negative or either offset or offset + count is greater than or equal to the size of this raw memory area.

Throws ArrayIndexOutOfBoundsException when start is negative or either start or start + count is greater than or equal to the size of values.

Throws NullPointerException when values is null.

```
@SCJAllowed
@SCJMayAllocate({})
@SCJMaySelfSuspend(false)
@SCJPhase({
    javax.safetycritical.annotate.Phase.STARTUP,
    javax.safetycritical.annotate.Phase.INITIALIZATION,
    javax.safetycritical.annotate.Phase.RUN,
    javax.safetycritical.annotate.Phase.CLEANUP })
public void setInt(int offset, int value)
  throws javax.realtime.OffsetOutOfBoundsException
```

Set the value of the n^{th} element referenced by this instance, where n is offset and the address is *base address* + offset x *size of Int*. This operation must be atomic with respect to all other raw memory accesses to the address. When an exception is thrown, no data is transfered.

offset — of int in the memory region.

value — is the new value for the element.

Throws OffsetOutOfBoundsException when offset is negative or greater than or equal to the number of elements in the raw memory region.

```
@SCJAllowed
@SCJMayAllocate({})
@SCJMaySelfSuspend(false)
@SCJPhase({
    javax.safetycritical.annotate.Phase.STARTUP,
```

javax.safetycritical.annotate.Phase.INITIALIZATION,
javax.safetycritical.annotate.Phase.RUN,
javax.safetycritical.annotate.Phase.CLEANUP })
public void setInt(**int** value)

Set the value at the first position referenced by this instance, i.e., the value at its start address. This operation must be atomic with respect to all other raw memory accesses to the address.

value — is the new value for the element.

E.2.13 INTERFACE **RawLong**

@SCJAllowed
public interface RawLong **extends** javax.realtime.device.RawLongReader,
javax.realtime.device.RawLongWriter

A marker for an object that can be used to access to a single long. Read and write access to that long is checked by the factory that creates the instance; therefore, no access checking is provided by this interface, only bounds checking.

Since
RTSJ 2.0

E.2.14 INTERFACE **RawLongReader**

@SCJAllowed
public interface RawLongReader **extends** javax.realtime.device.RawMemory

A marker for a long accessor object encapsulating the protocol for reading longs from raw memory. A long accessor can always access at least one long. Each long is transfered in a single atomic operation. Groups of longs may be transfered together; however, this is not required.

Objects of this type are created with the method **javax.realtime.device.RawMemoryFactorycreateRawLongReader** and **javax.realtime.device.RawMemoryFactorycreateRawLong** . Each object references a range of elements in the **javax.realtime.device.RawMemoryRegion** starting at the *base address* provided to the factory method. The size provided to the factor method determines the number of elements accessable.

Caching of the memory access is controlled by the factory that created this object. If the memory is not cached, this method guarantees serialized access. In other words, the memory access at the memory occurs in the same order as in the program. Multiple writes to the same location may not be coalesced.

Since
RTSJ 2.0

Methods

```
@SCJAllowed
@SCJMayAllocate({})
@SCJMaySelfSuspend(false)
@SCJPhase({
   javax.safetycritical.annotate.Phase.STARTUP,
   javax.safetycritical.annotate.Phase.INITIALIZATION,
   javax.safetycritical.annotate.Phase.RUN,
   javax.safetycritical.annotate.Phase.CLEANUP })
public int get(int offset, long [] values)
  throws javax.realtime.OffsetOutOfBoundsException,
     java.lang.NullPointerException
```

Fill values with elements from this instance, where the nth element is at the address: base address + (offset+n) x stride x element size in bytes. Only the longs in the intersection of the start and end of values and the *base address* and the end of the memory region are transfered. When an exception is thrown, no data is transfered.

offset — of the first long in the memory region to transfere

values — the array to receive the longs

returns the number of elements actuall transferred to values

Throws OffsetOutOfBoundsException when offset is negative or greater than or equal to the number of elements in the raw memory region.

Throws NullPointerException when values is null.

```
@SCJAllowed
@SCJMayAllocate({})
@SCJMaySelfSuspend(false)
@SCJPhase({
   javax.safetycritical.annotate.Phase.STARTUP,
   javax.safetycritical.annotate.Phase.INITIALIZATION,
   javax.safetycritical.annotate.Phase.RUN,
   javax.safetycritical.annotate.Phase.CLEANUP })
```

```
public int get(int offset, long [] values, int start, int count)
  throws javax.realtime.OffsetOutOfBoundsException,
     java.lang.ArrayIndexOutOfBoundsException, java.lang.NullPointerException
```

Fill values from index start with elements from this instance, where the nth element is at the address: base address + (offset+n) x stride x element size in bytes. The number of bytes transfered is the minimum of count, the *size* of the memory region minus offset, and length of values minus start. When an exception is thrown, no data is transfered.

offset — of the first long in the memory region to transfere

values — the array to receive the longs

start — the first index in array to fill

count — the maximum number of longs to copy

returns the number of longs actually transfered.

Throws OffsetOutOfBoundsException when offset is negative or either offset or offset + count is greater than or equal to the size of this raw memory area.

Throws ArrayIndexOutOfBoundsException when start is negative or either start or start + count is greater than or equal to the size of values.

Throws NullPointerException when values is null or count is negative.

```
@SCJAllowed
@SCJMayAllocate({})
@SCJMaySelfSuspend(false)
@SCJPhase({
    javax.safetycritical.annotate.Phase.STARTUP,
    javax.safetycritical.annotate.Phase.INITIALIZATION,
    javax.safetycritical.annotate.Phase.RUN,
    javax.safetycritical.annotate.Phase.CLEANUP })
public long getLong(int offset)
  throws javax.realtime.OffsetOutOfBoundsException
```

Get the value at the address: base address + offset x stride x element size in bytes. When an exception is thrown, no data is transfered.

offset — of long in the memory region starting from the address specified in the associated factory method.

returns the value at the address specified.

Throws OffsetOutOfBoundsException when offset is negative or greater than or equal to the number of elements in the raw memory region.

```
@SCJAllowed
@SCJMayAllocate({})
@SCJMaySelfSuspend(false)
@SCJPhase({
   javax.safetycritical.annotate.Phase.STARTUP,
   javax.safetycritical.annotate.Phase.INITIALIZATION,
   javax.safetycritical.annotate.Phase.RUN,
   javax.safetycritical.annotate.Phase.CLEANUP })
public long getLong( )
```

Get the value at the first position referenced by this instance, i.e., the value at its start address. This operation must be atomic with respect to all other raw memory accesses to the address.

returns the value at the *base address*.

E.2.15 INTERFACE **RawLongWriter**

```
@SCJAllowed
public interface RawLongWriter extends javax.realtime.device.RawMemory
```

A marker for a long accessor object encapsulating the protocol for writing longs to raw memory. A long accessor can always access at least one long. Each long is transfered in a single atomic operation. Groups of longs may be transfered together; however, this is not required.

Objects of this type are created with the method **javax.realtime.device.RawMemoryFactorycreateRawLongWriter** and **javax.realtime.device.RawMemoryFactorycreateRawLong** . Each object references a range of elements in the **javax.realtime.device.RawMemoryRegion** starting at the *base address* provided to the factory method. The size provided to the factor method determines the number of elements accessable.

Caching of the memory access is controlled by the factory that created this object. If the memory is not cached, this method guarantees serialized access. In other words, the memory access at the memory occurs in the same order as in the program. Multiple writes to the same location may not be coalesced.

Since
RTSJ 2.0

Methods

```
@SCJAllowed
@SCJMayAllocate({})
@SCJMaySelfSuspend(false)
@SCJPhase({
    javax.safetycritical.annotate.Phase.STARTUP,
    javax.safetycritical.annotate.Phase.INITIALIZATION,
    javax.safetycritical.annotate.Phase.RUN,
    javax.safetycritical.annotate.Phase.CLEANUP })
public int set(int offset, long [] values)
  throws javax.realtime.OffsetOutOfBoundsException,
      java.lang.NullPointerException
```

Copy from values to the memory region from index start,to elements where the nth element is at the address: base address + (offset+n) x stride x element size in bytes. Only the longs in the intersection of values and the end of the memory region are transfered. When an exception is thrown, no data is transfered.

offset — of first long in the memory region to be set.

values — is the source of the data to write.

returns the number of elements actually transferred to values

Throws OffsetOutOfBoundsException when offset is negative or greater than or equal to the number of elements in the raw memory region.

Throws NullPointerException when values is null.

```
@SCJAllowed
@SCJMayAllocate({})
@SCJMaySelfSuspend(false)
@SCJPhase({
    javax.safetycritical.annotate.Phase.STARTUP,
    javax.safetycritical.annotate.Phase.INITIALIZATION,
    javax.safetycritical.annotate.Phase.RUN,
    javax.safetycritical.annotate.Phase.CLEANUP })
public int set(int offset, long [] values, int start, int count)
  throws javax.realtime.OffsetOutOfBoundsException,
      java.lang.ArrayIndexOutOfBoundsException, java.lang.NullPointerException
```

Copy values to the memory region, where offset is first long in the memory region to write and start is the first index in values from which to read. The number of bytes transfered is the minimum of count, the *size* of the memory region minus offset, and length of values minus start. When an exception is thrown, no data is transfered.

offset — of the first long in the memory region to set

values — the array from which to retrieve the longs

start — the first index in array to copy

count — the maximum number of longs to copy

returns the number of longs actually transfered.

Throws OffsetOutOfBoundsException when offset is negative or either offset or offset + count is greater than or equal to the size of this raw memory area.

Throws ArrayIndexOutOfBoundsException when start is negative or either start or start + count is greater than or equal to the size of values.

Throws NullPointerException when values is null.

```
@SCJAllowed
@SCJMayAllocate({})
@SCJMaySelfSuspend(false)
@SCJPhase({
   javax.safetycritical.annotate.Phase.STARTUP,
   javax.safetycritical.annotate.Phase.INITIALIZATION,
   javax.safetycritical.annotate.Phase.RUN,
   javax.safetycritical.annotate.Phase.CLEANUP })
public void setLong(int offset, long value)
  throws javax.realtime.OffsetOutOfBoundsException
```

Set the value of the n^{th} element referenced by this instance, where n is offset and the address is *base address* + offset x *size of Long*. This operation must be atomic with respect to all other raw memory accesses to the address. When an exception is thrown, no data is transfered.

offset — of long in the memory region.

value — is the new value for the element.

Throws OffsetOutOfBoundsException when offset is negative or greater than or equal to the number of elements in the raw memory region.

```
@SCJAllowed
@SCJMayAllocate({})
@SCJMaySelfSuspend(false)
@SCJPhase({
   javax.safetycritical.annotate.Phase.STARTUP,
   javax.safetycritical.annotate.Phase.INITIALIZATION,
   javax.safetycritical.annotate.Phase.RUN,
   javax.safetycritical.annotate.Phase.CLEANUP })
public void setLong(long value)
```

Set the value at the first position referenced by this instance, i.e., the value at its start address. This operation must be atomic with respect to all other raw memory accesses to the address.

value — is the new value for the element.

E.2.16 INTERFACE **RawMemoryRegionFactory**

```
@SCJAllowed
public interface RawMemoryRegionFactory
```

A class to give an application the ability to provide support for a **javax.realtime.device.RawMemoryRegion** that is not already provided by the standard. An instance of this call can be registered with a **javax.realtime.device.RawMemoryFactory** and provides the object that that factory should return for a given RawMemoryRegion. It is responsible for checking all requests and throwing the proper exception when a request is invalid or the requester is not authorized to make the request.

Since
RTSJ 2.0

Methods

```
@SCJAllowed
@SCJMayAllocate({javax.safetycritical.annotate.AllocationContext.CURRENT})
@SCJMaySelfSuspend(false)
@SCJPhase({
    javax.safetycritical.annotate.Phase.STARTUP,
    javax.safetycritical.annotate.Phase.INITIALIZATION,
    javax.safetycritical.annotate.Phase.RUN,
    javax.safetycritical.annotate.Phase.CLEANUP })
public javax.realtime.device.RawByte createRawByte(long base, int count, int stride)
  throws java.lang.SecurityException, javax.realtime.OffsetOutOfBoundsException,
       javax.realtime.SizeOutOfBoundsException,
      javax.realtime.UnsupportedRawMemoryRegionException,
      javax.realtime.MemoryTypeConflictException
```

Create an instance of a class that implements **javax.realtime.device.RawByte** and accesses memory of **javax.realtime.device.RawMemoryRegionFactorygetRegion** in the address range described by base, stride, and count. The actual extent of the memory addressed by the object is stride x *size of RawByte* x count. The object is allocated in the current memory area of the calling thread.

base — The starting physical address accessible through the returned instance.

count — The number of memory elements accessible through the returned instance.

stride — The distance to the next element in mulitple of element count, where a value of 1 means the elements are adjacent in memory.

returns an object that implements **javax.realtime.device.RawByte** and supports access to the specified range in the memory region.

Throws IllegalArgumentException when base is negative, count is not greater than zero, or stride is not greater than zero.

Throws SecurityException when the caller does not have permissions to access the given memory region or the specified range of addresses.

Throws OffsetOutOfBoundsException when base is invalid.

Throws SizeOutOfBoundsException when the memory addressed by the object would extend into an invalid range of memory.

Throws MemoryTypeConflictException when base does not point to memory that matches the type served by this factory.

Since
RTSJ 2.0

```
@SCJAllowed
@SCJMayAllocate({javax.safetycritical.annotate.AllocationContext.CURRENT})
@SCJMaySelfSuspend(false)
@SCJPhase({
    javax.safetycritical.annotate.Phase.STARTUP,
    javax.safetycritical.annotate.Phase.INITIALIZATION,
    javax.safetycritical.annotate.Phase.RUN,
    javax.safetycritical.annotate.Phase.CLEANUP })
public javax.realtime.device.RawByteReader createRawByteReader(long base,
  int count,
  int stride)
  throws java.lang.SecurityException, javax.realtime.OffsetOutOfBoundsException,
       javax.realtime.SizeOutOfBoundsException,
      javax.realtime.UnsupportedRawMemoryRegionException,
      javax.realtime.MemoryTypeConflictException
```

Create an instance of a class that implements **javax.realtime.device.RawByteReader** and accesses memory of **javax.realtime.device.RawMemoryRegionFactorygetRegion** in the address range described by base, stride, and count. The actual extent of the memory addressed by the object is stride x *size of RawByteReader* x count. The object is allocated in the current memory area of the calling thread.

base — The starting physical address accessible through the returned instance.

count — The number of memory elements accessible through the returned instance.

stride — The distance to the next element in mulitple of element count, where a value of 1 means the elements are adjacent in memory.

returns an object that implements **javax.realtime.device.RawByteReader** and supports access to the specified range in the memory region.

Throws IllegalArgumentException when base is negative, count is not greater than zero, or stride is not greater than zero.

Throws SecurityException when the caller does not have permissions to access the given memory region or the specified range of addresses.

Throws OffsetOutOfBoundsException when base is invalid.

Throws SizeOutOfBoundsException when the memory addressed by the object would extend into an invalid range of memory.

Throws MemoryTypeConflictException when base does not point to memory that matches the type served by this factory.

Since
RTSJ 2.0

```
@SCJAllowed
@SCJMayAllocate({javax.safetycritical.annotate.AllocationContext.CURRENT})
@SCJMaySelfSuspend(false)
@SCJPhase({
    javax.safetycritical.annotate.Phase.STARTUP,
    javax.safetycritical.annotate.Phase.INITIALIZATION,
    javax.safetycritical.annotate.Phase.RUN,
    javax.safetycritical.annotate.Phase.CLEANUP })
public javax.realtime.device.RawByteWriter createRawByteWriter(long base,
  int count,
  int stride)
  throws java.lang.SecurityException, javax.realtime.OffsetOutOfBoundsException,
       javax.realtime.SizeOutOfBoundsException,
      javax.realtime.UnsupportedRawMemoryRegionException,
      javax.realtime.MemoryTypeConflictException
```

Create an instance of a class that implements **javax.realtime.device.RawByteWriter** and accesses memory of **javax.realtime.device.RawMemoryRegionFactorygetRegion** in the address range described by base, stride, and count. The actual extent of the memory addressed by the object is stride x *size of RawByteWriter* x count. The object is allocated in the current memory area of the calling thread.

base — The starting physical address accessible through the returned instance.

count — The number of memory elements accessible through the returned instance.

stride — The distance to the next element in mulitple of element count, where a value of 1 means the elements are adjacent in memory.

returns an object that implements **javax.realtime.device.RawByteWriter** and supports access to the specified range in the memory region.

Throws IllegalArgumentException when base is negative, count is not greater than zero, or stride is not greater than zero.

Throws SecurityException when the caller does not have permissions to access the given memory region or the specified range of addresses.

Throws OffsetOutOfBoundsException when base is invalid.

Throws SizeOutOfBoundsException when the memory addressed by the object would extend into an invalid range of memory.

Throws MemoryTypeConflictException when base does not point to memory that matches the type served by this factory.

Since
RTSJ 2.0

```
@SCJAllowed
@SCJMayAllocate({javax.safetycritical.annotate.AllocationContext.CURRENT})
@SCJMaySelfSuspend(false)
@SCJPhase({
    javax.safetycritical.annotate.Phase.STARTUP,
    javax.safetycritical.annotate.Phase.INITIALIZATION,
    javax.safetycritical.annotate.Phase.RUN,
    javax.safetycritical.annotate.Phase.CLEANUP })
public javax.realtime.device.RawDouble createRawDouble(long base,
  int count,
  int stride)
  throws java.lang.SecurityException, javax.realtime.OffsetOutOfBoundsException,
       javax.realtime.SizeOutOfBoundsException,
      javax.realtime.UnsupportedRawMemoryRegionException,
      javax.realtime.MemoryTypeConflictException
```

Create an instance of a class that implements **javax.realtime.device.RawDouble** and accesses memory of **javax.realtime.device.RawMemoryRegionFactoryget-Region** in the address range described by base, stride, and count. The actual extent of the memory addressed by the object is stride x *size of RawDouble* x count. The object is allocated in the current memory area of the calling thread.

base — The starting physical address accessible through the returned instance.

count — The number of memory elements accessible through the returned instance.

stride — The distance to the next element in mulitple of element count, where a value of 1 means the elements are adjacent in memory.

returns an object that implements **javax.realtime.device.RawDouble** and supports access to the specified range in the memory region.

Throws IllegalArgumentException when base is negative, count is not greater than zero, or stride is not greater than zero.

Throws SecurityException when the caller does not have permissions to access the given memory region or the specified range of addresses.

Throws OffsetOutOfBoundsException when base is invalid.

Throws SizeOutOfBoundsException when the memory addressed by the object would extend into an invalid range of memory.

Throws MemoryTypeConflictException when base does not point to memory that matches the type served by this factory.

Since
RTSJ 2.0

```
@SCJAllowed
@SCJMayAllocate({javax.safetycritical.annotate.AllocationContext.CURRENT})
@SCJMaySelfSuspend(false)
@SCJPhase({
    javax.safetycritical.annotate.Phase.STARTUP,
    javax.safetycritical.annotate.Phase.INITIALIZATION,
    javax.safetycritical.annotate.Phase.RUN,
    javax.safetycritical.annotate.Phase.CLEANUP })
public javax.realtime.device.RawDoubleReader createRawDoubleReader(long base,
  int count,
  int stride)
  throws java.lang.SecurityException, javax.realtime.OffsetOutOfBoundsException,
       javax.realtime.SizeOutOfBoundsException,
      javax.realtime.UnsupportedRawMemoryRegionException,
      javax.realtime.MemoryTypeConflictException
```

Create an instance of a class that implements **javax.realtime.device.RawDoubleReader** and accesses memory of **javax.realtime.device.RawMemoryRegionFactorygetRegion** in the address range described by base, stride, and count. The actual extent of the memory addressed by the object is stride x *size of RawDoubleReader* x count. The object is allocated in the current memory area of the calling thread.

base — The starting physical address accessible through the returned instance.

count — The number of memory elements accessible through the returned instance.

stride — The distance to the next element in mulitple of element count, where a value of 1 means the elements are adjacent in memory.

returns an object that implements **javax.realtime.device.RawDoubleReader** and supports access to the specified range in the memory region.

Throws IllegalArgumentException when base is negative, count is not greater than zero, or stride is not greater than zero.

Throws SecurityException when the caller does not have permissions to access the given memory region or the specified range of addresses.

Throws OffsetOutOfBoundsException when base is invalid.

Throws SizeOutOfBoundsException when the memory addressed by the object would extend into an invalid range of memory.

Throws MemoryTypeConflictException when base does not point to memory that matches the type served by this factory.

Since
RTSJ 2.0

```
@SCJAllowed
@SCJMayAllocate({javax.safetycritical.annotate.AllocationContext.CURRENT})
@SCJMaySelfSuspend(false)
@SCJPhase({
    javax.safetycritical.annotate.Phase.STARTUP,
    javax.safetycritical.annotate.Phase.INITIALIZATION,
    javax.safetycritical.annotate.Phase.RUN,
    javax.safetycritical.annotate.Phase.CLEANUP })
public javax.realtime.device.RawDoubleWriter createRawDoubleWriter(long base,
  int count,
  int stride)
  throws java.lang.SecurityException, javax.realtime.OffsetOutOfBoundsException,
       javax.realtime.SizeOutOfBoundsException,
      javax.realtime.UnsupportedRawMemoryRegionException,
      javax.realtime.MemoryTypeConflictException
```

Create an instance of a class that implements **javax.realtime.device.RawDoubleWriter** and accesses memory of **javax.realtime.device.RawMemoryRegionFactorygetRegion** in the address range described by base, stride, and count. The actual extent of the memory addressed by the object is stride x *size of RawDoubleWriter* x count. The object is allocated in the current memory area of the calling thread.

base — The starting physical address accessible through the returned instance.

count — The number of memory elements accessible through the returned instance.

stride — The distance to the next element in mulitple of element count, where a value of 1 means the elements are adjacent in memory.

returns an object that implements **javax.realtime.device.RawDoubleWriter** and supports access to the specified range in the memory region.

Throws IllegalArgumentException when base is negative, count is not greater than zero, or stride is not greater than zero.

Throws SecurityException when the caller does not have permissions to access the given memory region or the specified range of addresses.

Throws OffsetOutOfBoundsException when base is invalid.

Throws SizeOutOfBoundsException when the memory addressed by the object would extend into an invalid range of memory.

Throws MemoryTypeConflictException when base does not point to memory that matches the type served by this factory.

Since
RTSJ 2.0

```
@SCJAllowed
@SCJMayAllocate({javax.safetycritical.annotate.AllocationContext.CURRENT})
@SCJMaySelfSuspend(false)
@SCJPhase({
    javax.safetycritical.annotate.Phase.STARTUP,
    javax.safetycritical.annotate.Phase.INITIALIZATION,
    javax.safetycritical.annotate.Phase.RUN,
    javax.safetycritical.annotate.Phase.CLEANUP })
public javax.realtime.device.RawFloat createRawFloat(long base,
  int count,
  int stride)
  throws java.lang.SecurityException, javax.realtime.OffsetOutOfBoundsException,
      javax.realtime.SizeOutOfBoundsException,
     javax.realtime.UnsupportedRawMemoryRegionException,
     javax.realtime.MemoryTypeConflictException
```

Create an instance of a class that implements **javax.realtime.device.RawFloat** and accesses memory of **javax.realtime.device.RawMemoryRegionFactoryget-Region** in the address range described by base, stride, and count. The actual extent of the memory addressed by the object is stride x *size of RawFloat* x count. The object is allocated in the current memory area of the calling thread.

base — The starting physical address accessible through the returned instance.

count — The number of memory elements accessible through the returned instance.

stride — The distance to the next element in mulitple of element count, where a value of 1 means the elements are adjacent in memory.

returns an object that implements **javax.realtime.device.RawFloat** and supports access to the specified range in the memory region.

Throws IllegalArgumentException when base is negative, count is not greater than zero, or stride is not greater than zero.

Throws SecurityException when the caller does not have permissions to access the given memory region or the specified range of addresses.

Throws OffsetOutOfBoundsException when base is invalid.

Throws SizeOutOfBoundsException when the memory addressed by the object would extend into an invalid range of memory.

Throws MemoryTypeConflictException when base does not point to memory that matches the type served by this factory.

Since
RTSJ 2.0

```
@SCJAllowed
@SCJMayAllocate({javax.safetycritical.annotate.AllocationContext.CURRENT})
@SCJMaySelfSuspend(false)
@SCJPhase({
    javax.safetycritical.annotate.Phase.STARTUP,
    javax.safetycritical.annotate.Phase.INITIALIZATION,
    javax.safetycritical.annotate.Phase.RUN,
    javax.safetycritical.annotate.Phase.CLEANUP })
public javax.realtime.device.RawFloatReader createRawFloatReader(long base,
  int count,
  int stride)
  throws java.lang.SecurityException, javax.realtime.OffsetOutOfBoundsException,
       javax.realtime.SizeOutOfBoundsException,
      javax.realtime.UnsupportedRawMemoryRegionException,
      javax.realtime.MemoryTypeConflictException
```

Create an instance of a class that implements **javax.realtime.device.RawFloatReader** and accesses memory of **javax.realtime.device.RawMemoryRegionFactorygetRegion** in the address range described by base, stride, and count. The actual extent of the memory addressed by the object is stride x *size of RawFloatReader* x count. The object is allocated in the current memory area of the calling thread.

base — The starting physical address accessible through the returned instance.

count — The number of memory elements accessible through the returned instance.

stride — The distance to the next element in mulitple of element count, where a value of 1 means the elements are adjacent in memory.

returns an object that implements **javax.realtime.device.RawFloatReader** and supports access to the specified range in the memory region.

Throws IllegalArgumentException when base is negative, count is not greater than zero, or stride is not greater than zero.

Throws SecurityException when the caller does not have permissions to access the given memory region or the specified range of addresses.

Throws OffsetOutOfBoundsException when base is invalid.

Throws SizeOutOfBoundsException when the memory addressed by the object would extend into an invalid range of memory.

Throws MemoryTypeConflictException when base does not point to memory that matches the type served by this factory.

Since
RTSJ 2.0

```
@SCJAllowed
@SCJMayAllocate({javax.safetycritical.annotate.AllocationContext.CURRENT})
@SCJMaySelfSuspend(false)
@SCJPhase({
    javax.safetycritical.annotate.Phase.STARTUP,
    javax.safetycritical.annotate.Phase.INITIALIZATION,
    javax.safetycritical.annotate.Phase.RUN,
    javax.safetycritical.annotate.Phase.CLEANUP })
public javax.realtime.device.RawFloatWriter createRawFloatWriter(long base,
  int count,
  int stride)
  throws java.lang.SecurityException, javax.realtime.OffsetOutOfBoundsException,
       javax.realtime.SizeOutOfBoundsException,
      javax.realtime.UnsupportedRawMemoryRegionException,
      javax.realtime.MemoryTypeConflictException
```

Create an instance of a class that implements **javax.realtime.device.RawFloatWriter** and accesses memory of **javax.realtime.device.RawMemoryRegionFactorygetRegion** in the address range described by base, stride, and count. The actual extent of the memory addressed by the object is stride x *size of RawFloatWriter* x count. The object is allocated in the current memory area of the calling thread.

base — The starting physical address accessible through the returned instance.

count — The number of memory elements accessible through the returned instance.

stride — The distance to the next element in mulitple of element count, where a value of 1 means the elements are adjacent in memory.

returns an object that implements **javax.realtime.device.RawFloatWriter** and supports access to the specified range in the memory region.

Throws IllegalArgumentException when base is negative, count is not greater than zero, or stride is not greater than zero.

Throws SecurityException when the caller does not have permissions to access the given memory region or the specified range of addresses.

Throws OffsetOutOfBoundsException when base is invalid.

Throws SizeOutOfBoundsException when the memory addressed by the object would extend into an invalid range of memory.

Throws MemoryTypeConflictException when base does not point to memory that matches the type served by this factory.

Since
RTSJ 2.0

```
@SCJAllowed
@SCJMayAllocate({javax.safetycritical.annotate.AllocationContext.CURRENT})
@SCJMaySelfSuspend(false)
@SCJPhase({
    javax.safetycritical.annotate.Phase.STARTUP,
    javax.safetycritical.annotate.Phase.INITIALIZATION,
    javax.safetycritical.annotate.Phase.RUN,
    javax.safetycritical.annotate.Phase.CLEANUP })
public javax.realtime.device.RawInt createRawInt(long base, int count, int stride)
  throws java.lang.SecurityException, javax.realtime.OffsetOutOfBoundsException,
       javax.realtime.SizeOutOfBoundsException,
      javax.realtime.UnsupportedRawMemoryRegionException,
      javax.realtime.MemoryTypeConflictException
```

Create an instance of a class that implements **javax.realtime.device.RawInt** and accesses memory of **javax.realtime.device.RawMemoryRegionFactoryget-Region** in the address range described by base, stride, and count. The actual extent of the memory addressed by the object is stride x *size of RawInt* x count. The object is allocated in the current memory area of the calling thread.

base — The starting physical address accessible through the returned instance.

count — The number of memory elements accessible through the returned instance.

stride — The distance to the next element in mulitple of element count, where a value of 1 means the elements are adjacent in memory.

returns an object that implements **javax.realtime.device.RawInt** and supports access to the specified range in the memory region.

Throws IllegalArgumentException when base is negative, count is not greater than zero, or stride is not greater than zero.

Throws SecurityException when the caller does not have permissions to access the given memory region or the specified range of addresses.

Throws OffsetOutOfBoundsException when base is invalid.

Throws SizeOutOfBoundsException when the memory addressed by the object would extend into an invalid range of memory.

Throws MemoryTypeConflictException when base does not point to memory that matches the type served by this factory.

Since
RTSJ 2.0

```
@SCJAllowed
@SCJMayAllocate({javax.safetycritical.annotate.AllocationContext.CURRENT})
@SCJMaySelfSuspend(false)
@SCJPhase({
    javax.safetycritical.annotate.Phase.STARTUP,
    javax.safetycritical.annotate.Phase.INITIALIZATION,
    javax.safetycritical.annotate.Phase.RUN,
    javax.safetycritical.annotate.Phase.CLEANUP })
public javax.realtime.device.RawIntReader createRawIntReader(long base,
  int count,
  int stride)
  throws java.lang.SecurityException, javax.realtime.OffsetOutOfBoundsException,
       javax.realtime.SizeOutOfBoundsException,
      javax.realtime.UnsupportedRawMemoryRegionException,
      javax.realtime.MemoryTypeConflictException
```

Create an instance of a class that implements **javax.realtime.device.RawIntReader** and accesses memory of **javax.realtime.device.RawMemoryRegionFactorygetRegion** in the address range described by base, stride, and count. The actual extent of the memory addressed by the object is stride x *size of RawIntReader* x count. The object is allocated in the current memory area of the calling thread.

base — The starting physical address accessible through the returned instance.

count — The number of memory elements accessible through the returned instance.

stride — The distance to the next element in mulitple of element count, where a value of 1 means the elements are adjacent in memory.

returns an object that implements **javax.realtime.device.RawIntReader** and supports access to the specified range in the memory region.

Throws IllegalArgumentException when base is negative, count is not greater than zero, or stride is not greater than zero.

Throws SecurityException when the caller does not have permissions to access the given memory region or the specified range of addresses.

Throws OffsetOutOfBoundsException when base is invalid.

Throws SizeOutOfBoundsException when the memory addressed by the object would extend into an invalid range of memory.

Throws MemoryTypeConflictException when base does not point to memory that matches the type served by this factory.

Since
RTSJ 2.0

```
@SCJAllowed
@SCJMayAllocate({javax.safetycritical.annotate.AllocationContext.CURRENT})
@SCJMaySelfSuspend(false)
@SCJPhase({
    javax.safetycritical.annotate.Phase.STARTUP,
    javax.safetycritical.annotate.Phase.INITIALIZATION,
    javax.safetycritical.annotate.Phase.RUN,
    javax.safetycritical.annotate.Phase.CLEANUP })
public javax.realtime.device.RawIntWriter createRawIntWriter(long base,
  int count,
  int stride)
  throws java.lang.SecurityException, javax.realtime.OffsetOutOfBoundsException,
       javax.realtime.SizeOutOfBoundsException,
      javax.realtime.UnsupportedRawMemoryRegionException,
      javax.realtime.MemoryTypeConflictException
```

Create an instance of a class that implements **javax.realtime.device.RawIntWriter** and accesses memory of **javax.realtime.device.RawMemoryRegionFactorygetRegion** in the address range described by base, stride, and count. The actual extent of the memory addressed by the object is stride x *size of RawIntWriter* x count. The object is allocated in the current memory area of the calling thread.

base — The starting physical address accessible through the returned instance.

count — The number of memory elements accessible through the returned instance.

stride — The distance to the next element in mulitple of element count, where a value of 1 means the elements are adjacent in memory.

returns an object that implements **javax.realtime.device.RawIntWriter** and supports access to the specified range in the memory region.

Throws IllegalArgumentException when base is negative, count is not greater than zero, or stride is not greater than zero.

Throws SecurityException when the caller does not have permissions to access the given memory region or the specified range of addresses.

Throws OffsetOutOfBoundsException when base is invalid.

Throws SizeOutOfBoundsException when the memory addressed by the object would extend into an invalid range of memory.

Throws MemoryTypeConflictException when base does not point to memory that matches the type served by this factory.

Since
RTSJ 2.0

```
@SCJAllowed
@SCJMayAllocate({javax.safetycritical.annotate.AllocationContext.CURRENT})
@SCJMaySelfSuspend(false)
@SCJPhase({
    javax.safetycritical.annotate.Phase.STARTUP,
    javax.safetycritical.annotate.Phase.INITIALIZATION,
    javax.safetycritical.annotate.Phase.RUN,
    javax.safetycritical.annotate.Phase.CLEANUP })
public javax.realtime.device.RawLong createRawLong(long base, int count, int stride)
  throws java.lang.SecurityException, javax.realtime.OffsetOutOfBoundsException,
       javax.realtime.SizeOutOfBoundsException,
      javax.realtime.UnsupportedRawMemoryRegionException,
      javax.realtime.MemoryTypeConflictException
```

Create an instance of a class that implements **javax.realtime.device.RawLong** and accesses memory of **javax.realtime.device.RawMemoryRegionFactoryget-Region** in the address range described by base, stride, and count. The actual extent of the memory addressed by the object is stride x *size of RawLong* x count. The object is allocated in the current memory area of the calling thread.

base — The starting physical address accessible through the returned instance.

count — The number of memory elements accessible through the returned instance.

stride — The distance to the next element in mulitple of element count, where a value of 1 means the elements are adjacent in memory.

returns an object that implements **javax.realtime.device.RawLong** and supports access to the specified range in the memory region.

Throws IllegalArgumentException when base is negative, count is not greater than zero, or stride is not greater than zero.

Throws SecurityException when the caller does not have permissions to access the given memory region or the specified range of addresses.

Throws OffsetOutOfBoundsException when base is invalid.

Throws SizeOutOfBoundsException when the memory addressed by the object would extend into an invalid range of memory.

Throws MemoryTypeConflictException when base does not point to memory that matches the type served by this factory.

Since
RTSJ 2.0

```
@SCJAllowed
@SCJMayAllocate({javax.safetycritical.annotate.AllocationContext.CURRENT})
@SCJMaySelfSuspend(false)
@SCJPhase({
    javax.safetycritical.annotate.Phase.STARTUP,
    javax.safetycritical.annotate.Phase.INITIALIZATION,
    javax.safetycritical.annotate.Phase.RUN,
    javax.safetycritical.annotate.Phase.CLEANUP })
public javax.realtime.device.RawLongReader createRawLongReader(long base,
  int count,
  int stride)
  throws java.lang.SecurityException, javax.realtime.OffsetOutOfBoundsException,
       javax.realtime.SizeOutOfBoundsException,
      javax.realtime.UnsupportedRawMemoryRegionException,
      javax.realtime.MemoryTypeConflictException
```

Create an instance of a class that implements **javax.realtime.device.RawLongReader** and accesses memory of **javax.realtime.device.RawMemoryRegionFactorygetRegion** in the address range described by base, stride, and count. The actual extent of the memory addressed by the object is stride x *size of RawLongReader* x count. The object is allocated in the current memory area of the calling thread.

base — The starting physical address accessible through the returned instance.

count — The number of memory elements accessible through the returned instance.

stride — The distance to the next element in mulitple of element count, where a value of 1 means the elements are adjacent in memory.

returns an object that implements **javax.realtime.device.RawLongReader** and supports access to the specified range in the memory region.

Throws IllegalArgumentException when base is negative, count is not greater than zero, or stride is not greater than zero.

Throws SecurityException when the caller does not have permissions to access the given memory region or the specified range of addresses.

Throws OffsetOutOfBoundsException when base is invalid.

Throws SizeOutOfBoundsException when the memory addressed by the object would extend into an invalid range of memory.

Throws MemoryTypeConflictException when base does not point to memory that matches the type served by this factory.

Since
RTSJ 2.0

```
@SCJAllowed
@SCJMayAllocate({javax.safetycritical.annotate.AllocationContext.CURRENT})
@SCJMaySelfSuspend(false)
@SCJPhase({
    javax.safetycritical.annotate.Phase.STARTUP,
    javax.safetycritical.annotate.Phase.INITIALIZATION,
    javax.safetycritical.annotate.Phase.RUN,
    javax.safetycritical.annotate.Phase.CLEANUP })
public javax.realtime.device.RawLongWriter createRawLongWriter(long base,
  int count,
  int stride)
  throws java.lang.SecurityException, javax.realtime.OffsetOutOfBoundsException,
       javax.realtime.SizeOutOfBoundsException,
      javax.realtime.UnsupportedRawMemoryRegionException,
      javax.realtime.MemoryTypeConflictException
```

Create an instance of a class that implements **javax.realtime.device.RawLongWriter** and accesses memory of **javax.realtime.device.RawMemoryRegionFactorygetRegion** in the address range described by base, stride, and count. The actual extent of the memory addressed by the object is stride x *size of RawLongWriter* x count. The object is allocated in the current memory area of the calling thread.

base — The starting physical address accessible through the returned instance.

count — The number of memory elements accessible through the returned instance.

stride — The distance to the next element in mulitple of element count, where a value of 1 means the elements are adjacent in memory.

returns an object that implements **javax.realtime.device.RawLongWriter** and supports access to the specified range in the memory region.

Throws IllegalArgumentException when base is negative, count is not greater than zero, or stride is not greater than zero.

Throws SecurityException when the caller does not have permissions to access the given memory region or the specified range of addresses.

Throws OffsetOutOfBoundsException when base is invalid.

Throws SizeOutOfBoundsException when the memory addressed by the object would extend into an invalid range of memory.

Throws MemoryTypeConflictException when base does not point to memory that matches the type served by this factory.

Since
RTSJ 2.0

```
@SCJAllowed
@SCJMayAllocate({javax.safetycritical.annotate.AllocationContext.CURRENT})
@SCJMaySelfSuspend(false)
@SCJPhase({
    javax.safetycritical.annotate.Phase.STARTUP,
    javax.safetycritical.annotate.Phase.INITIALIZATION,
    javax.safetycritical.annotate.Phase.RUN,
    javax.safetycritical.annotate.Phase.CLEANUP })
public javax.realtime.device.RawShort createRawShort(long base,
  int count,
  int stride)
  throws java.lang.SecurityException, javax.realtime.OffsetOutOfBoundsException,
       javax.realtime.SizeOutOfBoundsException,
      javax.realtime.UnsupportedRawMemoryRegionException,
      javax.realtime.MemoryTypeConflictException
```

Create an instance of a class that implements **javax.realtime.device.RawShort** and accesses memory of **javax.realtime.device.RawMemoryRegionFactoryget-Region** in the address range described by base, stride, and count. The actual extent of the memory addressed by the object is stride x *size of RawShort* x count. The object is allocated in the current memory area of the calling thread.

base — The starting physical address accessible through the returned instance.

count — The number of memory elements accessible through the returned instance.

stride — The distance to the next element in mulitple of element count, where a value of 1 means the elements are adjacent in memory.

returns an object that implements **javax.realtime.device.RawShort** and supports access to the specified range in the memory region.

Throws IllegalArgumentException when base is negative, count is not greater than zero, or stride is not greater than zero.

Throws SecurityException when the caller does not have permissions to access the given memory region or the specified range of addresses.

Throws OffsetOutOfBoundsException when base is invalid.

Throws SizeOutOfBoundsException when the memory addressed by the object would extend into an invalid range of memory.

Throws MemoryTypeConflictException when base does not point to memory that matches the type served by this factory.

Since
RTSJ 2.0

```
@SCJAllowed
@SCJMayAllocate({javax.safetycritical.annotate.AllocationContext.CURRENT})
@SCJMaySelfSuspend(false)
@SCJPhase({
    javax.safetycritical.annotate.Phase.STARTUP,
    javax.safetycritical.annotate.Phase.INITIALIZATION,
    javax.safetycritical.annotate.Phase.RUN,
    javax.safetycritical.annotate.Phase.CLEANUP })
public javax.realtime.device.RawShortReader createRawShortReader(long base,
  int count,
  int stride)
  throws java.lang.SecurityException, javax.realtime.OffsetOutOfBoundsException,
       javax.realtime.SizeOutOfBoundsException,
      javax.realtime.UnsupportedRawMemoryRegionException,
      javax.realtime.MemoryTypeConflictException
```

Create an instance of a class that implements **javax.realtime.device.RawShortReader** and accesses memory of **javax.realtime.device.RawMemoryRegionFactorygetRegion** in the address range described by base, stride, and count. The actual extent of the memory addressed by the object is stride x *size of RawShortReader* x count. The object is allocated in the current memory area of the calling thread.

base — The starting physical address accessible through the returned instance.

count — The number of memory elements accessible through the returned instance.

stride — The distance to the next element in mulitple of element count, where a value of 1 means the elements are adjacent in memory.

returns an object that implements **javax.realtime.device.RawShortReader** and supports access to the specified range in the memory region.

Throws IllegalArgumentException when base is negative, count is not greater than zero, or stride is not greater than zero.

Throws SecurityException when the caller does not have permissions to access the given memory region or the specified range of addresses.

Throws OffsetOutOfBoundsException when base is invalid.

Throws SizeOutOfBoundsException when the memory addressed by the object would extend into an invalid range of memory.

Throws MemoryTypeConflictException when base does not point to memory that matches the type served by this factory.

Since
RTSJ 2.0

```
@SCJAllowed
@SCJMayAllocate({javax.safetycritical.annotate.AllocationContext.CURRENT})
@SCJMaySelfSuspend(false)
@SCJPhase({
    javax.safetycritical.annotate.Phase.STARTUP,
    javax.safetycritical.annotate.Phase.INITIALIZATION,
    javax.safetycritical.annotate.Phase.RUN,
    javax.safetycritical.annotate.Phase.CLEANUP })
public javax.realtime.device.RawShortWriter createRawShortWriter(long base,
  int count,
  int stride)
  throws java.lang.SecurityException, javax.realtime.OffsetOutOfBoundsException,
       javax.realtime.SizeOutOfBoundsException,
      javax.realtime.UnsupportedRawMemoryRegionException,
      javax.realtime.MemoryTypeConflictException
```

Create an instance of a class that implements **javax.realtime.device.RawShortWriter** and accesses memory of **javax.realtime.device.RawMemoryRegionFactorygetRegion** in the address range described by base, stride, and count. The actual extent of the memory addressed by the object is stride x *size of RawShortWriter* x count. The object is allocated in the current memory area of the calling thread.

base — The starting physical address accessible through the returned instance.

count — The number of memory elements accessible through the returned instance.

stride — The distance to the next element in mulitple of element count, where a value of 1 means the elements are adjacent in memory.

returns an object that implements **javax.realtime.device.RawShortWriter** and supports access to the specified range in the memory region.

Throws IllegalArgumentException when base is negative, count is not greater than zero, or stride is not greater than zero.

Throws SecurityException when the caller does not have permissions to access the given memory region or the specified range of addresses.

Throws OffsetOutOfBoundsException when base is invalid.

Throws SizeOutOfBoundsException when the memory addressed by the object would extend into an invalid range of memory.

Throws MemoryTypeConflictException when base does not point to memory that matches the type served by this factory.

Since
RTSJ 2.0

```
@SCJAllowed
@SCJMayAllocate({})
@SCJMaySelfSuspend(false)
@SCJPhase({
    javax.safetycritical.annotate.Phase.STARTUP,
    javax.safetycritical.annotate.Phase.INITIALIZATION,
    javax.safetycritical.annotate.Phase.RUN,
    javax.safetycritical.annotate.Phase.CLEANUP })
public java.lang.String getName( )
```

Determine the name of the region for which this factory creates raw memory objects.

returns the name of the region of this factory.

```
@SCJAllowed
@SCJMayAllocate({})
@SCJMaySelfSuspend(false)
@SCJPhase({
    javax.safetycritical.annotate.Phase.STARTUP,
    javax.safetycritical.annotate.Phase.INITIALIZATION,
    javax.safetycritical.annotate.Phase.RUN,
    javax.safetycritical.annotate.Phase.CLEANUP })
public javax.realtime.device.RawMemoryRegion getRegion( )
```

Determine for what region this factory creates raw memory objects.

returns the region of this factory.

E.2.17 INTERFACE **RawShort**

```
@SCJAllowed
public interface RawShort extends javax.realtime.device.RawShortReader,
    javax.realtime.device.RawShortWriter
```

A marker for an object that can be used to access to a single short. Read and write access to that short is checked by the factory that creates the instance; therefore, no access checking is provided by this interface, only bounds checking.

Since
RTSJ 2.0

E.2.18 INTERFACE **RawShortReader**

@SCJAllowed
public interface RawShortReader **extends** javax.realtime.device.RawMemory

A marker for a short accessor object encapsulating the protocol for reading shorts from raw memory. A short accessor can always access at least one short. Each short is transfered in a single atomic operation. Groups of shorts may be transfered together; however, this is not required.

Objects of this type are created with the method **javax.realtime.device.RawMemoryFactorycreateRawShortReader** and **javax.realtime.device.RawMemoryFactorycreateRawShort** . Each object references a range of elements in the **javax.realtime.device.RawMemoryRegion** starting at the *base address* provided to the factory method. The size provided to the factor method determines the number of elements accessable.

Caching of the memory access is controlled by the factory that created this object. If the memory is not cached, this method guarantees serialized access. In other words, the memory access at the memory occurs in the same order as in the program. Multiple writes to the same location may not be coalesced.

Since
RTSJ 2.0

Methods

```
@SCJAllowed
@SCJMayAllocate({})
@SCJMaySelfSuspend(false)
@SCJPhase({
   javax.safetycritical.annotate.Phase.STARTUP,
   javax.safetycritical.annotate.Phase.INITIALIZATION,
   javax.safetycritical.annotate.Phase.RUN,
   javax.safetycritical.annotate.Phase.CLEANUP })
```

```
public int get(int offset, short [] values)
  throws javax.realtime.OffsetOutOfBoundsException,
     java.lang.NullPointerException
```

Fill values with elements from this instance, where the nth element is at the address: base address + (offset+n) x stride x element size in bytes. Only the shorts in the intersection of the start and end of values and the *base address* and the end of the memory region are transfered. When an exception is thrown, no data is transfered.

offset — of the first short in the memory region to transfere

values — the array to receive the shorts

returns the number of elements actuall transferred to values

Throws OffsetOutOfBoundsException when offset is negative or greater than or equal to the number of elements in the raw memory region.

Throws NullPointerException when values is null.

```
@SCJAllowed
@SCJMayAllocate({})
@SCJMaySelfSuspend(false)
@SCJPhase({
    javax.safetycritical.annotate.Phase.STARTUP,
    javax.safetycritical.annotate.Phase.INITIALIZATION,
    javax.safetycritical.annotate.Phase.RUN,
    javax.safetycritical.annotate.Phase.CLEANUP })
public int get(int offset, short [] values, int start, int count)
  throws javax.realtime.OffsetOutOfBoundsException,
     java.lang.ArrayIndexOutOfBoundsException, java.lang.NullPointerException
```

Fill values from index start with elements from this instance, where the nth element is at the address: base address + (offset+n) x stride x element size in bytes. The number of bytes transfered is the minimum of count, the *size* of the memory region minus offset, and length of values minus start. When an exception is thrown, no data is transfered.

offset — of the first short in the memory region to transfere

values — the array to receive the shorts

start — the first index in array to fill

count — the maximum number of shorts to copy

returns the number of shorts actually transfered.

Throws OffsetOutOfBoundsException when offset is negative or either offset or offset + count is greater than or equal to the size of this raw memory area.

Throws ArrayIndexOutOfBoundsException when start is negative or either start or start + count is greater than or equal to the size of values.

Throws NullPointerException when values is null or count is negative.

```
@SCJAllowed
@SCJMayAllocate({})
@SCJMaySelfSuspend(false)
@SCJPhase({
    javax.safetycritical.annotate.Phase.STARTUP,
    javax.safetycritical.annotate.Phase.INITIALIZATION,
    javax.safetycritical.annotate.Phase.RUN,
    javax.safetycritical.annotate.Phase.CLEANUP })
public short getShort(int offset)
  throws javax.realtime.OffsetOutOfBoundsException
```

Get the value at the address: base address + offset x stride x element size in bytes. When an exception is thrown, no data is transfered.

offset — of short in the memory region starting from the address specified in the associated factory method.

returns the value at the address specified.

Throws OffsetOutOfBoundsException when offset is negative or greater than or equal to the number of elements in the raw memory region.

```
@SCJAllowed
@SCJMayAllocate({})
@SCJMaySelfSuspend(false)
@SCJPhase({
    javax.safetycritical.annotate.Phase.STARTUP,
    javax.safetycritical.annotate.Phase.INITIALIZATION,
    javax.safetycritical.annotate.Phase.RUN,
    javax.safetycritical.annotate.Phase.CLEANUP })
public short getShort( )
```

Get the value at the first position referenced by this instance, i.e., the value at its start address. This operation must be atomic with respect to all other raw memory accesses to the address.

returns the value at the *base address*.

E.2.19 INTERFACE **RawShortWriter**

```
@SCJAllowed
public interface RawShortWriter extends javax.realtime.device.RawMemory
```

A marker for a short accessor object encapsulating the protocol for writing shorts to raw memory. A short accessor can always access at least one short. Each short is transfered in a single atomic operation. Groups of shorts may be transfered together; however, this is not required.

Objects of this type are created with the method **javax.realtime.device.RawMemoryFactorycreateRawShortWriter** and **javax.realtime.device.RawMemoryFactorycreateRawShort** . Each object references a range of elements in the **javax.realtime.device.RawMemoryRegion** starting at the *base address* provided to the factory method. The size provided to the factor method determines the number of elements accessable.

Caching of the memory access is controlled by the factory that created this object. If the memory is not cached, this method guarantees serialized access. In other words, the memory access at the memory occurs in the same order as in the program. Multiple writes to the same location may not be coalesced.

Since
RTSJ 2.0

Methods

```
@SCJAllowed
@SCJMayAllocate({})
@SCJMaySelfSuspend(false)
@SCJPhase({
    javax.safetycritical.annotate.Phase.STARTUP,
    javax.safetycritical.annotate.Phase.INITIALIZATION,
    javax.safetycritical.annotate.Phase.RUN,
    javax.safetycritical.annotate.Phase.CLEANUP })
public int set(int offset, short [] values)
  throws javax.realtime.OffsetOutOfBoundsException,
      java.lang.NullPointerException
```

Copy from values to the memory region from index start,to elements where the nth element is at the address: base address + (offset+n) x stride x element size in bytes. Only the shorts in the intersection of values and the end of the memory region are transfered. When an exception is thrown, no data is transfered.

offset — of first short in the memory region to be set.

values — is the source of the data to write.

returns the number of elements actually transferred to values

Throws OffsetOutOfBoundsException when offset is negative or greater than or equal to the number of elements in the raw memory region.

Throws NullPointerException when values is null.

```
@SCJAllowed
@SCJMayAllocate({})
@SCJMaySelfSuspend(false)
@SCJPhase({
    javax.safetycritical.annotate.Phase.STARTUP,
    javax.safetycritical.annotate.Phase.INITIALIZATION,
    javax.safetycritical.annotate.Phase.RUN,
    javax.safetycritical.annotate.Phase.CLEANUP })
public int set(int offset, short [] values, int start, int count)
  throws javax.realtime.OffsetOutOfBoundsException,
      java.lang.ArrayIndexOutOfBoundsException, java.lang.NullPointerException
```

Copy values to the memory region, where offset is first short in the memory region to write and start is the first index in values from which to read. The number of bytes transfered is the minimum of count, the *size* of the memory region minus offset, and length of values minus start. When an exception is thrown, no data is transfered.

offset — of the first short in the memory region to set

values — the array from which to retrieve the shorts

start — the first index in array to copy

count — the maximum number of shorts to copy

returns the number of shorts actually transfered.

Throws OffsetOutOfBoundsException when offset is negative or either offset or offset + count is greater than or equal to the size of this raw memory area.

Throws ArrayIndexOutOfBoundsException when start is negative or either start or start + count is greater than or equal to the size of values.

Throws NullPointerException when values is null.

```
@SCJAllowed
@SCJMayAllocate({})
@SCJMaySelfSuspend(false)
@SCJPhase({
    javax.safetycritical.annotate.Phase.STARTUP,
    javax.safetycritical.annotate.Phase.INITIALIZATION,
    javax.safetycritical.annotate.Phase.RUN,
    javax.safetycritical.annotate.Phase.CLEANUP })
public void setShort(int offset, short value)
  throws javax.realtime.OffsetOutOfBoundsException
```

Set the value of the n^{th} element referenced by this instance, where n is offset and the address is *base address* + offset x *size of Short*. This operation must be atomic with respect to all other raw memory accesses to the address. When an exception is thrown, no data is transfered.

offset — of short in the memory region.

value — is the new value for the element.

Throws OffsetOutOfBoundsException when offset is negative or greater than or equal to the number of elements in the raw memory region.

```
@SCJAllowed
@SCJMayAllocate({})
@SCJMaySelfSuspend(false)
@SCJPhase({
    javax.safetycritical.annotate.Phase.STARTUP,
    javax.safetycritical.annotate.Phase.INITIALIZATION,
    javax.safetycritical.annotate.Phase.RUN,
    javax.safetycritical.annotate.Phase.CLEANUP })
public void setShort(short value)
```

Set the value at the first position referenced by this instance, i.e., the value at its start address. This operation must be atomic with respect to all other raw memory accesses to the address.

value — is the new value for the element.

E.3 Classes

E.3.1 CLASS InterruptServiceRoutine

```
@SCJAllowed(javax.safetycritical.annotate.Level.LEVEL_1)
public abstract class InterruptServiceRoutine implements
    javax.realtime.BoundRealtimeExecutor extends java.lang.Object
```

A first level interrupt handling mechanisms. Override the handle method to provide the first level interrupt handler. The constructors for this class are invoked by the infrastructure and are therefore not visible to the application. The default affinity of an handler can be determined via calling **getAffinity()** .

Methods

```
@SCJAllowed(javax.safetycritical.annotate.Level.LEVEL_1)
@SCJPhase({
    javax.safetycritical.annotate.Phase.INITIALIZATION,
    javax.safetycritical.annotate.Phase.RUN,
    javax.safetycritical.annotate.Phase.CLEANUP})
@SCJMaySelfSuspend(false)
@SCJMayAllocate({
    javax.safetycritical.annotate.AllocationContext.CURRENT,
    javax.safetycritical.annotate.AllocationContext.INNER,
    javax.safetycritical.annotate.AllocationContext.OUTER})
public static javax.realtime.device.InterruptServiceRoutine getHandler(
  int interrupt)
```

Find the InterruptServiceRoutine that is handling a given interrupt.

interrupt — for which to find the InterruptServiceRoutine

returns the InterruptServiceRoutine registered to the given interrupt. Null is returned when nothing is registered for that interrupt.

```
@SCJAllowed(javax.safetycritical.annotate.Level.LEVEL_1)
@SCJMayAllocate({})
@SCJMaySelfSuspend(false)
@SCJPhase({
    javax.safetycritical.annotate.Phase.STARTUP,
    javax.safetycritical.annotate.Phase.INITIALIZATION,
    javax.safetycritical.annotate.Phase.RUN,
    javax.safetycritical.annotate.Phase.CLEANUP })
public static int getInterruptPriority(int InterruptId)
```

Every interrupt has an implementation-defined integer id.

returns The priority of the code that the first-level interrupts code executes. The returned value is always greater than PriorityScheduler.getMaxPriority().

Throws IllegalArgumentException if unsupported InterruptId

```
@SCJAllowed(javax.safetycritical.annotate.Level.LEVEL_1)
@SCJPhase({
    javax.safetycritical.annotate.Phase.INITIALIZATION,
    javax.safetycritical.annotate.Phase.RUN,
    javax.safetycritical.annotate.Phase.CLEANUP})
@SCJMaySelfSuspend(false)
@SCJMayAllocate({
    javax.safetycritical.annotate.AllocationContext.CURRENT,
    javax.safetycritical.annotate.AllocationContext.INNER,
    javax.safetycritical.annotate.AllocationContext.OUTER})
public static int getMaximumInterruptPriority( )
```

Retrieve the maximum interrupt priority. It must be greater than or equal to the result of getMinimumInterruptPriority.

returns the maximum interrupt priority.

```
@SCJAllowed(javax.safetycritical.annotate.Level.LEVEL_1)
@SCJPhase({
   javax.safetycritical.annotate.Phase.INITIALIZATION,
   javax.safetycritical.annotate.Phase.RUN,
   javax.safetycritical.annotate.Phase.CLEANUP})
@SCJMaySelfSuspend(false)
@SCJMayAllocate({
   javax.safetycritical.annotate.AllocationContext.CURRENT,
   javax.safetycritical.annotate.AllocationContext.INNER,
   javax.safetycritical.annotate.AllocationContext.OUTER})
public static int getMinimumInterruptPriority( )
```

Retrieve the minimum interrupt priority. It must be higher than all other priorities provided by the system.

returns the minimum interrupt priority.

```
@SCJAllowed(javax.safetycritical.annotate.Level.LEVEL_1)
@SCJPhase({
   javax.safetycritical.annotate.Phase.INITIALIZATION,
   javax.safetycritical.annotate.Phase.RUN,
   javax.safetycritical.annotate.Phase.CLEANUP})
@SCJMaySelfSuspend(false)
@SCJMayAllocate({javax.safetycritical.annotate.AllocationContext.CURRENT})
protected abstract void handle( )
```

The code to execute for first level interrupt handling. A subclass defines this to give the proper behavior. No code that could self-suspend may be called here. The effects of unbound blocking and inducing a context switch here are undefined and could result in deadlocking the machine. Unless the overridden method is synchronized, the infrastructure shall provide no synchronization for the execution of this method.

E.3.2 CLASS **RawMemoryFactory**

```
@SCJAllowed
public class RawMemoryFactory extends java.lang.Object
```

This class is the hub of a system that constructs special purpose objects to access particular types and ranges of raw memory. This facility is supported by the RawMemoryRegionFactory methods. An application developer can use this method to add support for additional memory regions.

Each create method returns an object of the corresponding type, e.g., the createRawByte(RawMemoryRegion, long, int, int)} method returns a reference to an object that implements the **javax.realtime.device.RawByte** interface and supports access to the requested type of memory and address range. Each create method is permitted to optimize error checking and access based on the requested memory type and address range.

The usage pattern for raw memory, assuming the necessary factory has been registered, is illustrated by this example.

```
// Get an accessor object that can access memory starting at
// baseAddress, for size bytes.
RawInt memory =
  RawMemoryFactory.createRawInt(RawMemoryFactory.MEMORY_MAPPED_REGION,
                                address, count, stride, false);
// Use the accessor to load from and store to raw memory.
int loadedData = memory.getInt(someOffset);
memory.setInt(otherOffset, intVal);
```

When an application needs to access a class of memory that is not already supported by a registered factory, the developer must implement and register a factory that implements the **javax.realtime.device.RawMemoryRegionFactory**) which can create objects to access memory in that region.

A raw memory region factory is identified by a **javax.realtime.device.RawMemoryRegion** that is used by each create method, e.g., **createRawByte(RawMemoryRegion, long, int, int)** , to locate the appropriate factory. The name is provided to **register(RawMemoryRegionFactory)** through the factory's **javax.realtime.device.RawMemoryRegionFactorygetName** method.

The **register(RawMemoryRegionFactory)** method is only used when by application code when it needs to add support for a new type of raw memory.

Whether a give offset addresses a high-order or low-order byte of an aligned short in memory is determined by the value of the javax.realtime.RealtimeSystem.BYTE_ORDER static byte variable in class javax.realtime.RealtimeSystem, the start address of the object, and the offset given the stride of the object. Regardless of the byte ordering, accessor methods continue to select bytes starting at offset from the base address and continuing toward greater addresses.

A raw memory region cannot contain references to Java objects. Such a capability would be unsafe (since it could be used to defeat Java's type checking)

and error prone (since it is sensitive to the specific representational choices made by the Java compiler).

Atomic loads and stores on raw memory are defined in terms of physical memory. This memory may be accessible to threads outside the JVM and to non-programmed access (e.g., DMA). Consequently, atomic access must be supported by hardware. This specification is written with the assumption that all suitable hardware platforms support atomic loads from raw memory for aligned bytes, shorts, and ints. Atomic access beyond the specified minimum may be supported by the implementation.

Storing values into raw memory is more hardware-dependent than loading values. Many processor architectures do not support atomic stores of variables except for aligned stores of the processor's word size. For instance, storing a byte into memory might require reading a 32-bit quantity into a processor register, updating the register to reflect the new byte value, then restoring the whole 32-bit quantity. Changes to other bytes in the 32-bit quantity that take place between the load and the store are lost.

Some processors have mechanisms that can be used to implement an atomic store of a byte, but those mechanisms are often slow and not universally supported.

This class need not support unaligned access to data; but if it does, it is not require the implementation to make such access atomic. Accesses to data aligned on its natural boundary will be atomic if the processor implements atomic loads and stores of that data size.

Except where noted, accesses to raw memory are not atomic with respect to the memory or with respect to schedulable objects. A raw memory region could be updated by another schedulable object, or even unmapped in the middle of an access method, or even *removed* mid method.

The characteristics of raw-memory access are necessarily platform dependent. This specification provides a minimum requirement for the SCJ platform, but it also supports optional system properties that identify a platform's level of support for atomic raw put and get. The properties represent a four-dimensional sparse array of access type, data type, alignment, and atomicity with boolean values indicating whether that combination of access attributes is atomic. The default value for array entries is false. The permissable values of these array entries are:

- Access type - possible values are *read* and *write*.
- Data type - possible values are *byte*, *short*, *int*, *long*, *float*, and *double*.
- Alignment - possible values are 0 through 7, inclusive. For each data type, the possible alignments range from:

- 0 means aligned
- 1 to (data size-1) means only the first byte of the data is alignment bytes away from natural alignment

- Atomicity - possible values are *processor*, *smp*, and *memory*.
  - *processor* means that access is atomic with respect to other schedulable objects on that processor.
  - *smp* means that access is processor atomic, and atomic across the processors in an SMP.
  - *memory* means that access is SMP atomic, and atomic with respect to all access to the memory, including DMA hardware.

The true values in the table are represented by properties of the following form. javax.realtime.atomicaccess_<access>_<type>_<alignment>_atomicity=true for example,

```
javax.realtime.atomicaccess_read_byte_0_memory=true
```

Table entries with a value of false may be explicitly represented, but since false is the default value, such properties are redundant.

All raw memory access is treated as volatile, and *serialized*. The infrastructure must be forced to read memory or write to memory on each call to a raw memory objects's getter or setter method, and to complete the reads and writes in the order they appear in the program order.

Since
RTSJ 2.0

Fields

@SCJAllowed
public static final javax.realtime.device.RawMemoryRegion IO_PORT_MAPPED_REGION

This raw memory region is predefined for access to I/O device space implemented by processor instructions, such as the x86 in and out instructions.

@SCJAllowed
public static final javax.realtime.device.RawMemoryRegion MEMORY_MAPPED_REGION

This raw memory region is predefined for request access to memory mapped I/O devices.

Constructors

```
@SCJAllowed
@SCJMaySelfSuspend(false)
@SCJPhase({
    javax.safetycritical.annotate.Phase.STARTUP,
    javax.safetycritical.annotate.Phase.INITIALIZATION})
@SCJMayAllocate({
    javax.safetycritical.annotate.AllocationContext.CURRENT,
    javax.safetycritical.annotate.AllocationContext.INNER,
    javax.safetycritical.annotate.AllocationContext.OUTER})
public RawMemoryFactory( )
```

Create an empty factory. For a factory with support for the platform defined regions, use **javax.realtime.device.RawMemoryFactorygetDefaultFactory** instead.

Methods

```
@SCJAllowed
@SCJMayAllocate({javax.safetycritical.annotate.AllocationContext.CURRENT})
@SCJMaySelfSuspend(false)
@SCJPhase({
    javax.safetycritical.annotate.Phase.STARTUP,
    javax.safetycritical.annotate.Phase.INITIALIZATION,
    javax.safetycritical.annotate.Phase.RUN,
    javax.safetycritical.annotate.Phase.CLEANUP })
public javax.realtime.device.RawByte createRawByte(RawMemoryRegion region,
  long base,
  int count,
  int stride)
  throws java.lang.SecurityException, javax.realtime.OffsetOutOfBoundsException,
      javax.realtime.SizeOutOfBoundsException,
     javax.realtime.MemoryTypeConflictException,
     javax.realtime.UnsupportedRawMemoryRegionException
```

Create an instance of a class that implements **javax.realtime.device.RawByte** and accesses memory of region in the address range described by base, stride, and count. The actual extent of the memory addressed by the object is stride x *size of RawByte* x count. The object is allocated in the current memory area of the calling thread.

region — The address space from which the new instance should be taken.

base — The starting physical address accessible through the returned instance.

count — The number of memory elements accessible through the returned instance.

stride — The distance to the next element as a mulitple of element size, where 1 means the elements are adjacent in memory.

returns an object that implements **javax.realtime.device.RawByte** and supports access to the specified range in the memory region.

Throws IllegalArgumentException when base is negative, count is not greater than zero, or stride is less than one.

Throws SecurityException when the caller does not have permissions to access the given memory region or the specified range of addresses.

Throws OffsetOutOfBoundsException when base is invalid.

Throws SizeOutOfBoundsException when the memory addressed by the object would extend into an invalid range of memory.

Throws MemoryTypeConflictException when base does not point to a memory that matches the type served by this factory.

```
@SCJAllowed
@SCJMayAllocate({javax.safetycritical.annotate.AllocationContext.CURRENT})
@SCJMaySelfSuspend(false)
@SCJPhase({
    javax.safetycritical.annotate.Phase.STARTUP,
    javax.safetycritical.annotate.Phase.INITIALIZATION,
    javax.safetycritical.annotate.Phase.RUN,
    javax.safetycritical.annotate.Phase.CLEANUP })
public javax.realtime.device.RawByteReader createRawByteReader(
  RawMemoryRegion region,
  long base,
  int count,
  int stride)
  throws java.lang.SecurityException, javax.realtime.OffsetOutOfBoundsException,
       javax.realtime.SizeOutOfBoundsException,
      javax.realtime.MemoryTypeConflictException,
      javax.realtime.UnsupportedRawMemoryRegionException
```

Create an instance of a class that implements **javax.realtime.device.RawByte-Reader** and accesses memory of region in the address range described by base, stride, and count. The actual extent of the memory addressed by the object is stride x *size of RawByteReader* x count. The object is allocated in the current memory area of the calling thread.

region — The address space from which the new instance should be taken.

base — The starting physical address accessible through the returned instance.

count — The number of memory elements accessible through the returned instance.

stride — The distance to the next element as a mulitple of element size, where 1 means the elements are adjacent in memory.

returns an object that implements **javax.realtime.device.RawByteReader** and supports access to the specified range in the memory region.

Throws IllegalArgumentException when base is negative, count is not greater than zero, or stride is less than one.

Throws SecurityException when the caller does not have permissions to access the given memory region or the specified range of addresses.

Throws OffsetOutOfBoundsException when base is invalid.

Throws SizeOutOfBoundsException when the memory addressed by the object would extend into an invalid range of memory.

Throws MemoryTypeConflictException when base does not point to a memory that matches the type served by this factory.

```
@SCJAllowed
@SCJMayAllocate({javax.safetycritical.annotate.AllocationContext.CURRENT})
@SCJMaySelfSuspend(false)
@SCJPhase({
    javax.safetycritical.annotate.Phase.STARTUP,
    javax.safetycritical.annotate.Phase.INITIALIZATION,
    javax.safetycritical.annotate.Phase.RUN,
    javax.safetycritical.annotate.Phase.CLEANUP })
public javax.realtime.device.RawByteWriter createRawByteWriter(
  RawMemoryRegion region,
  long base,
  int count,
  int stride)
  throws java.lang.SecurityException, javax.realtime.OffsetOutOfBoundsException,
       javax.realtime.SizeOutOfBoundsException,
      javax.realtime.MemoryTypeConflictException,
      javax.realtime.UnsupportedRawMemoryRegionException
```

Create an instance of a class that implements **javax.realtime.device.RawByteWriter** and accesses memory of region in the address range described by base, stride, and count. The actual extent of the memory addressed by the object is stride x *size of RawByteWriter* x count. The object is allocated in the current memory area of the calling thread.

region — The address space from which the new instance should be taken.

base — The starting physical address accessible through the returned instance.

count — The number of memory elements accessible through the returned instance.

stride — The distance to the next element as a mulitple of element size, where 1 means the elements are adjacent in memory.

returns an object that implements **javax.realtime.device.RawByteWriter** and supports access to the specified range in the memory region.

Throws IllegalArgumentException when base is negative, count is not greater than zero, or stride is less than one.

Throws SecurityException when the caller does not have permissions to access the given memory region or the specified range of addresses.

Throws OffsetOutOfBoundsException when base is invalid.

Throws SizeOutOfBoundsException when the memory addressed by the object would extend into an invalid range of memory.

Throws MemoryTypeConflictException when base does not point to a memory that matches the type served by this factory.

```
@SCJAllowed
@SCJMayAllocate({javax.safetycritical.annotate.AllocationContext.CURRENT})
@SCJMaySelfSuspend(false)
@SCJPhase({
    javax.safetycritical.annotate.Phase.STARTUP,
    javax.safetycritical.annotate.Phase.INITIALIZATION,
    javax.safetycritical.annotate.Phase.RUN,
    javax.safetycritical.annotate.Phase.CLEANUP })
public javax.realtime.device.RawDouble createRawDouble(RawMemoryRegion region,
  long base,
  int count,
  int stride)
  throws java.lang.SecurityException, javax.realtime.OffsetOutOfBoundsException,
       javax.realtime.SizeOutOfBoundsException,
      javax.realtime.MemoryTypeConflictException,
      javax.realtime.UnsupportedRawMemoryRegionException
```

Create an instance of a class that implements **javax.realtime.device.RawDouble** and accesses memory of region in the address range described by base, stride, and count. The actual extent of the memory addressed by the object is stride x *size of RawDouble* x count. The object is allocated in the current memory area of the calling thread.

region — The address space from which the new instance should be taken.

base — The starting physical address accessible through the returned instance.

count — The number of memory elements accessible through the returned instance.

stride — The distance to the next element as a mulitple of element size, where 1 means the elements are adjacent in memory.

returns an object that implements **javax.realtime.device.RawDouble** and supports access to the specified range in the memory region.

Throws IllegalArgumentException when base is negative, count is not greater than zero, or stride is less than one.

Throws SecurityException when the caller does not have permissions to access the given memory region or the specified range of addresses.

Throws OffsetOutOfBoundsException when base is invalid.

Throws SizeOutOfBoundsException when the memory addressed by the object would extend into an invalid range of memory.

Throws MemoryTypeConflictException when base does not point to a memory that matches the type served by this factory.

```
@SCJAllowed
@SCJMayAllocate({javax.safetycritical.annotate.AllocationContext.CURRENT})
@SCJMaySelfSuspend(false)
@SCJPhase({
    javax.safetycritical.annotate.Phase.STARTUP,
    javax.safetycritical.annotate.Phase.INITIALIZATION,
    javax.safetycritical.annotate.Phase.RUN,
    javax.safetycritical.annotate.Phase.CLEANUP })
public javax.realtime.device.RawDoubleReader createRawDoubleReader(
  RawMemoryRegion region,
  long base,
  int count,
  int stride)
  throws java.lang.SecurityException, javax.realtime.OffsetOutOfBoundsException,
       javax.realtime.SizeOutOfBoundsException,
      javax.realtime.MemoryTypeConflictException,
      javax.realtime.UnsupportedRawMemoryRegionException
```

Create an instance of a class that implements **javax.realtime.device.RawDoubleReader** and accesses memory of region in the address range described by base, stride, and count. The actual extent of the memory addressed by the object is stride x *size of RawDoubleReader* x count. The object is allocated in the current memory area of the calling thread.

region — The address space from which the new instance should be taken.

base — The starting physical address accessible through the returned instance.

count — The number of memory elements accessible through the returned instance.

stride — The distance to the next element as a mulitple of element size, where 1 means the elements are adjacent in memory.

returns an object that implements **javax.realtime.device.RawDoubleReader** and supports access to the specified range in the memory region.

Throws IllegalArgumentException when base is negative, count is not greater than zero, or stride is less than one.

Throws SecurityException when the caller does not have permissions to access the given memory region or the specified range of addresses.

Throws OffsetOutOfBoundsException when base is invalid.

Throws SizeOutOfBoundsException when the memory addressed by the object would extend into an invalid range of memory.

Throws MemoryTypeConflictException when base does not point to a memory that matches the type served by this factory.

```
@SCJAllowed
@SCJMayAllocate({javax.safetycritical.annotate.AllocationContext.CURRENT})
@SCJMaySelfSuspend(false)
@SCJPhase({
    javax.safetycritical.annotate.Phase.STARTUP,
    javax.safetycritical.annotate.Phase.INITIALIZATION,
    javax.safetycritical.annotate.Phase.RUN,
    javax.safetycritical.annotate.Phase.CLEANUP })
public javax.realtime.device.RawDoubleWriter createRawDoubleWriter(
  RawMemoryRegion region,
  long base,
  int count,
  int stride)
  throws java.lang.SecurityException, javax.realtime.OffsetOutOfBoundsException,
       javax.realtime.SizeOutOfBoundsException,
      javax.realtime.MemoryTypeConflictException,
      javax.realtime.UnsupportedRawMemoryRegionException
```

Create an instance of a class that implements **javax.realtime.device.RawDoubleWriter** and accesses memory of region in the address range described by base, stride, and count. The actual extent of the memory addressed by the object is stride x *size of RawDoubleWriter* x count. The object is allocated in the current memory area of the calling thread.

region — The address space from which the new instance should be taken.

base — The starting physical address accessible through the returned instance.

count — The number of memory elements accessible through the returned instance.

stride — The distance to the next element as a mulitple of element size, where 1 means the elements are adjacent in memory.

returns an object that implements **javax.realtime.device.RawDoubleWriter** and supports access to the specified range in the memory region.

Throws IllegalArgumentException when base is negative, count is not greater than zero, or stride is less than one.

Throws SecurityException when the caller does not have permissions to access the given memory region or the specified range of addresses.

Throws OffsetOutOfBoundsException when base is invalid.

Throws SizeOutOfBoundsException when the memory addressed by the object would extend into an invalid range of memory.

Throws MemoryTypeConflictException when base does not point to a memory that matches the type served by this factory.

```
@SCJAllowed
@SCJMayAllocate({javax.safetycritical.annotate.AllocationContext.CURRENT})
@SCJMaySelfSuspend(false)
@SCJPhase({
    javax.safetycritical.annotate.Phase.STARTUP,
    javax.safetycritical.annotate.Phase.INITIALIZATION,
    javax.safetycritical.annotate.Phase.RUN,
    javax.safetycritical.annotate.Phase.CLEANUP })
public javax.realtime.device.RawFloat createRawFloat(RawMemoryRegion region,
  long base,
  int count,
  int stride)
  throws java.lang.SecurityException, javax.realtime.OffsetOutOfBoundsException,
       javax.realtime.SizeOutOfBoundsException,
      javax.realtime.MemoryTypeConflictException,
      javax.realtime.UnsupportedRawMemoryRegionException
```

Create an instance of a class that implements **javax.realtime.device.RawFloat** and accesses memory of region in the address range described by base, stride, and count. The actual extent of the memory addressed by the object is stride x *size of RawFloat* x count. The object is allocated in the current memory area of the calling thread.

region — The address space from which the new instance should be taken.

base — The starting physical address accessible through the returned instance.

count — The number of memory elements accessible through the returned instance.

stride — The distance to the next element as a mulitple of element size, where 1 means the elements are adjacent in memory.

returns an object that implements **javax.realtime.device.RawFloat** and supports access to the specified range in the memory region.

Throws IllegalArgumentException when base is negative, count is not greater than zero, or stride is less than one.

Throws SecurityException when the caller does not have permissions to access the given memory region or the specified range of addresses.

Throws OffsetOutOfBoundsException when base is invalid.

Throws SizeOutOfBoundsException when the memory addressed by the object would extend into an invalid range of memory.

Throws MemoryTypeConflictException when base does not point to a memory that matches the type served by this factory.

```
@SCJAllowed
@SCJMayAllocate({javax.safetycritical.annotate.AllocationContext.CURRENT})
@SCJMaySelfSuspend(false)
@SCJPhase({
    javax.safetycritical.annotate.Phase.STARTUP,
    javax.safetycritical.annotate.Phase.INITIALIZATION,
    javax.safetycritical.annotate.Phase.RUN,
    javax.safetycritical.annotate.Phase.CLEANUP })
public javax.realtime.device.RawFloatReader createRawFloatReader(
  RawMemoryRegion region,
  long base,
  int count,
  int stride)
  throws java.lang.SecurityException, javax.realtime.OffsetOutOfBoundsException,
       javax.realtime.SizeOutOfBoundsException,
      javax.realtime.MemoryTypeConflictException,
      javax.realtime.UnsupportedRawMemoryRegionException
```

Create an instance of a class that implements **javax.realtime.device.RawFloatReader** and accesses memory of region in the address range described by base, stride, and count. The actual extent of the memory addressed by the object is stride x *size of RawFloatReader* x count. The object is allocated in the current memory area of the calling thread.

region — The address space from which the new instance should be taken.

base — The starting physical address accessible through the returned instance.

count — The number of memory elements accessible through the returned instance.

stride — The distance to the next element as a mulitple of element size, where 1 means the elements are adjacent in memory.

returns an object that implements **javax.realtime.device.RawFloatReader** and supports access to the specified range in the memory region.

Throws IllegalArgumentException when base is negative, count is not greater than zero, or stride is less than one.

Throws SecurityException when the caller does not have permissions to access the given memory region or the specified range of addresses.

Throws OffsetOutOfBoundsException when base is invalid.

Throws SizeOutOfBoundsException when the memory addressed by the object would extend into an invalid range of memory.

Throws MemoryTypeConflictException when base does not point to a memory that matches the type served by this factory.

```
@SCJAllowed
@SCJMayAllocate({javax.safetycritical.annotate.AllocationContext.CURRENT})
@SCJMaySelfSuspend(false)
@SCJPhase({
    javax.safetycritical.annotate.Phase.STARTUP,
    javax.safetycritical.annotate.Phase.INITIALIZATION,
    javax.safetycritical.annotate.Phase.RUN,
    javax.safetycritical.annotate.Phase.CLEANUP })
public javax.realtime.device.RawFloatWriter createRawFloatWriter(
  RawMemoryRegion region,
  long base,
  int count,
  int stride)
  throws java.lang.SecurityException, javax.realtime.OffsetOutOfBoundsException,
       javax.realtime.SizeOutOfBoundsException,
      javax.realtime.MemoryTypeConflictException,
      javax.realtime.UnsupportedRawMemoryRegionException
```

Create an instance of a class that implements **javax.realtime.device.RawFloatWriter** and accesses memory of region in the address range described by base, stride, and count. The actual extent of the memory addressed by the object is stride x *size of RawFloatWriter* x count. The object is allocated in the current memory area of the calling thread.

region — The address space from which the new instance should be taken.

base — The starting physical address accessible through the returned instance.

count — The number of memory elements accessible through the returned instance.

stride — The distance to the next element as a mulitple of element size, where 1 means the elements are adjacent in memory.

returns an object that implements **javax.realtime.device.RawFloatWriter** and supports access to the specified range in the memory region.

Throws IllegalArgumentException when base is negative, count is not greater than zero, or stride is less than one.

Throws SecurityException when the caller does not have permissions to access the given memory region or the specified range of addresses.

Throws OffsetOutOfBoundsException when base is invalid.

Throws SizeOutOfBoundsException when the memory addressed by the object would extend into an invalid range of memory.

Throws MemoryTypeConflictException when base does not point to a memory that matches the type served by this factory.

```
@SCJAllowed
@SCJMayAllocate({javax.safetycritical.annotate.AllocationContext.CURRENT})
@SCJMaySelfSuspend(false)
@SCJPhase({
    javax.safetycritical.annotate.Phase.STARTUP,
    javax.safetycritical.annotate.Phase.INITIALIZATION,
    javax.safetycritical.annotate.Phase.RUN,
    javax.safetycritical.annotate.Phase.CLEANUP })
public javax.realtime.device.RawInt createRawInt(RawMemoryRegion region,
  long base,
  int count,
  int stride)
  throws java.lang.SecurityException, javax.realtime.OffsetOutOfBoundsException,
       javax.realtime.SizeOutOfBoundsException,
      javax.realtime.MemoryTypeConflictException,
      javax.realtime.UnsupportedRawMemoryRegionException
```

Create an instance of a class that implements **javax.realtime.device.RawInt** and accesses memory of region in the address range described by base, stride, and count. The actual extent of the memory addressed by the object is stride x *size of RawInt* x count. The object is allocated in the current memory area of the calling thread.

region — The address space from which the new instance should be taken.

base — The starting physical address accessible through the returned instance.

count — The number of memory elements accessible through the returned instance.

stride — The distance to the next element as a mulitple of element size, where 1 means the elements are adjacent in memory.

returns an object that implements **javax.realtime.device.RawInt** and supports access to the specified range in the memory region.

Throws IllegalArgumentException when base is negative, count is not greater than zero, or stride is less than one.

Throws SecurityException when the caller does not have permissions to access the given memory region or the specified range of addresses.

Throws OffsetOutOfBoundsException when base is invalid.

Throws SizeOutOfBoundsException when the memory addressed by the object would extend into an invalid range of memory.

Throws MemoryTypeConflictException when base does not point to a memory that matches the type served by this factory.

```
@SCJAllowed
@SCJMayAllocate({javax.safetycritical.annotate.AllocationContext.CURRENT})
@SCJMaySelfSuspend(false)
@SCJPhase({
    javax.safetycritical.annotate.Phase.STARTUP,
    javax.safetycritical.annotate.Phase.INITIALIZATION,
    javax.safetycritical.annotate.Phase.RUN,
    javax.safetycritical.annotate.Phase.CLEANUP })
public javax.realtime.device.RawIntReader createRawIntReader(
  RawMemoryRegion region,
  long base,
  int count,
  int stride)
  throws java.lang.SecurityException, javax.realtime.OffsetOutOfBoundsException,
       javax.realtime.SizeOutOfBoundsException,
      javax.realtime.MemoryTypeConflictException,
      javax.realtime.UnsupportedRawMemoryRegionException
```

Create an instance of a class that implements **javax.realtime.device.RawIntReader** and accesses memory of region in the address range described by base, stride, and count. The actual extent of the memory addressed by the object is stride x *size of RawIntReader* x count. The object is allocated in the current memory area of the calling thread.

region — The address space from which the new instance should be taken.

base — The starting physical address accessible through the returned instance.

count — The number of memory elements accessible through the returned instance.

stride — The distance to the next element as a mulitple of element size, where 1 means the elements are adjacent in memory.

returns an object that implements **javax.realtime.device.RawIntReader** and supports access to the specified range in the memory region.

Throws IllegalArgumentException when base is negative, count is not greater than zero, or stride is less than one.

Throws SecurityException when the caller does not have permissions to access the given memory region or the specified range of addresses.

Throws OffsetOutOfBoundsException when base is invalid.

Throws SizeOutOfBoundsException when the memory addressed by the object would extend into an invalid range of memory.

Throws MemoryTypeConflictException when base does not point to a memory that matches the type served by this factory.

```
@SCJAllowed
@SCJMayAllocate({javax.safetycritical.annotate.AllocationContext.CURRENT})
@SCJMaySelfSuspend(false)
@SCJPhase({
    javax.safetycritical.annotate.Phase.STARTUP,
    javax.safetycritical.annotate.Phase.INITIALIZATION,
    javax.safetycritical.annotate.Phase.RUN,
    javax.safetycritical.annotate.Phase.CLEANUP })
public javax.realtime.device.RawIntWriter createRawIntWriter(
  RawMemoryRegion region,
  long base,
  int count,
  int stride)
  throws java.lang.SecurityException, javax.realtime.OffsetOutOfBoundsException,
       javax.realtime.SizeOutOfBoundsException,
      javax.realtime.MemoryTypeConflictException,
      javax.realtime.UnsupportedRawMemoryRegionException
```

Create an instance of a class that implements **javax.realtime.device.RawIntWriter** and accesses memory of region in the address range described by base, stride, and count. The actual extent of the memory addressed by the object is stride x *size of RawIntWriter* x count. The object is allocated in the current memory area of the calling thread.

region — The address space from which the new instance should be taken.

base — The starting physical address accessible through the returned instance.

count — The number of memory elements accessible through the returned instance.

stride — The distance to the next element as a mulitple of element size, where 1 means the elements are adjacent in memory.

returns an object that implements **javax.realtime.device.RawIntWriter** and supports access to the specified range in the memory region.

Throws IllegalArgumentException when base is negative, count is not greater than zero, or stride is less than one.

Throws SecurityException when the caller does not have permissions to access the given memory region or the specified range of addresses.

Throws OffsetOutOfBoundsException when base is invalid.

Throws SizeOutOfBoundsException when the memory addressed by the object would extend into an invalid range of memory.

Throws MemoryTypeConflictException when base does not point to a memory that matches the type served by this factory.

```
@SCJAllowed
@SCJMayAllocate({javax.safetycritical.annotate.AllocationContext.CURRENT})
@SCJMaySelfSuspend(false)
@SCJPhase({
    javax.safetycritical.annotate.Phase.STARTUP,
    javax.safetycritical.annotate.Phase.INITIALIZATION,
    javax.safetycritical.annotate.Phase.RUN,
    javax.safetycritical.annotate.Phase.CLEANUP })
public javax.realtime.device.RawLong createRawLong(RawMemoryRegion region,
  long base,
  int count,
  int stride)
  throws java.lang.SecurityException, javax.realtime.OffsetOutOfBoundsException,
       javax.realtime.SizeOutOfBoundsException,
      javax.realtime.MemoryTypeConflictException,
      javax.realtime.UnsupportedRawMemoryRegionException
```

Create an instance of a class that implements **javax.realtime.device.RawLong** and accesses memory of region in the address range described by base, stride, and count. The actual extent of the memory addressed by the object is stride x *size of RawLong* x count. The object is allocated in the current memory area of the calling thread.

region — The address space from which the new instance should be taken.

base — The starting physical address accessible through the returned instance.

count — The number of memory elements accessible through the returned instance.

stride — The distance to the next element as a mulitple of element size, where 1 means the elements are adjacent in memory.

returns an object that implements **javax.realtime.device.RawLong** and supports access to the specified range in the memory region.

Throws IllegalArgumentException when base is negative, count is not greater than zero, or stride is less than one.

Throws SecurityException when the caller does not have permissions to access the given memory region or the specified range of addresses.

Throws OffsetOutOfBoundsException when base is invalid.

Throws SizeOutOfBoundsException when the memory addressed by the object would extend into an invalid range of memory.

Throws MemoryTypeConflictException when base does not point to a memory that matches the type served by this factory.

```
@SCJAllowed
@SCJMayAllocate({javax.safetycritical.annotate.AllocationContext.CURRENT})
@SCJMaySelfSuspend(false)
@SCJPhase({
    javax.safetycritical.annotate.Phase.STARTUP,
    javax.safetycritical.annotate.Phase.INITIALIZATION,
    javax.safetycritical.annotate.Phase.RUN,
    javax.safetycritical.annotate.Phase.CLEANUP })
public javax.realtime.device.RawLongReader createRawLongReader(
  RawMemoryRegion region,
  long base,
  int count,
  int stride)
  throws java.lang.SecurityException, javax.realtime.OffsetOutOfBoundsException,
       javax.realtime.SizeOutOfBoundsException,
      javax.realtime.MemoryTypeConflictException,
      javax.realtime.UnsupportedRawMemoryRegionException
```

Create an instance of a class that implements **javax.realtime.device.RawLongReader** and accesses memory of region in the address range described by base, stride, and count. The actual extent of the memory addressed by the object is stride x *size of RawLongReader* x count. The object is allocated in the current memory area of the calling thread.

region — The address space from which the new instance should be taken.

base — The starting physical address accessible through the returned instance.

count — The number of memory elements accessible through the returned instance.

stride — The distance to the next element as a mulitple of element size, where 1 means the elements are adjacent in memory.

returns an object that implements **javax.realtime.device.RawLongReader** and supports access to the specified range in the memory region.

Throws IllegalArgumentException when base is negative, count is not greater than zero, or stride is less than one.

Throws SecurityException when the caller does not have permissions to access the given memory region or the specified range of addresses.

Throws OffsetOutOfBoundsException when base is invalid.

Throws SizeOutOfBoundsException when the memory addressed by the object would extend into an invalid range of memory.

Throws MemoryTypeConflictException when base does not point to a memory that matches the type served by this factory.

```
@SCJAllowed
@SCJMayAllocate({javax.safetycritical.annotate.AllocationContext.CURRENT})
@SCJMaySelfSuspend(false)
@SCJPhase({
    javax.safetycritical.annotate.Phase.STARTUP,
    javax.safetycritical.annotate.Phase.INITIALIZATION,
    javax.safetycritical.annotate.Phase.RUN,
    javax.safetycritical.annotate.Phase.CLEANUP })
public javax.realtime.device.RawLongWriter createRawLongWriter(
  RawMemoryRegion region,
  long base,
  int count,
  int stride)
  throws java.lang.SecurityException, javax.realtime.OffsetOutOfBoundsException,
      javax.realtime.SizeOutOfBoundsException,
     javax.realtime.MemoryTypeConflictException,
     javax.realtime.UnsupportedRawMemoryRegionException
```

Create an instance of a class that implements **javax.realtime.device.RawLongWriter** and accesses memory of region in the address range described by base, stride, and count. The actual extent of the memory addressed by the object is stride x *size of RawLongWriter* x count. The object is allocated in the current memory area of the calling thread.

region — The address space from which the new instance should be taken.

base — The starting physical address accessible through the returned instance.

count — The number of memory elements accessible through the returned instance.

stride — The distance to the next element as a mulitple of element size, where 1 means the elements are adjacent in memory.

returns an object that implements **javax.realtime.device.RawLongWriter** and supports access to the specified range in the memory region.

Throws IllegalArgumentException when base is negative, count is not greater than zero, or stride is less than one.

Throws SecurityException when the caller does not have permissions to access the given memory region or the specified range of addresses.

Throws OffsetOutOfBoundsException when base is invalid.

Throws SizeOutOfBoundsException when the memory addressed by the object would extend into an invalid range of memory.

Throws MemoryTypeConflictException when base does not point to a memory that matches the type served by this factory.

```
@SCJAllowed
@SCJMayAllocate({javax.safetycritical.annotate.AllocationContext.CURRENT})
@SCJMaySelfSuspend(false)
@SCJPhase({
    javax.safetycritical.annotate.Phase.STARTUP,
    javax.safetycritical.annotate.Phase.INITIALIZATION,
    javax.safetycritical.annotate.Phase.RUN,
    javax.safetycritical.annotate.Phase.CLEANUP })
public javax.realtime.device.RawShort createRawShort(RawMemoryRegion region,
  long base,
  int count,
  int stride)
  throws java.lang.SecurityException, javax.realtime.OffsetOutOfBoundsException,
       javax.realtime.SizeOutOfBoundsException,
      javax.realtime.MemoryTypeConflictException,
      javax.realtime.UnsupportedRawMemoryRegionException
```

Create an instance of a class that implements **javax.realtime.device.RawShort** and accesses memory of region in the address range described by base, stride, and count. The actual extent of the memory addressed by the object is stride x *size of RawShort* x count. The object is allocated in the current memory area of the calling thread.

region — The address space from which the new instance should be taken.

base — The starting physical address accessible through the returned instance.

count — The number of memory elements accessible through the returned instance.

stride — The distance to the next element as a mulitple of element size, where 1 means the elements are adjacent in memory.

returns an object that implements **javax.realtime.device.RawShort** and supports access to the specified range in the memory region.

Throws IllegalArgumentException when base is negative, count is not greater than zero, or stride is less than one.

Throws SecurityException when the caller does not have permissions to access the given memory region or the specified range of addresses.

Throws OffsetOutOfBoundsException when base is invalid.

Throws SizeOutOfBoundsException when the memory addressed by the object would extend into an invalid range of memory.

Throws MemoryTypeConflictException when base does not point to a memory that matches the type served by this factory.

```
@SCJAllowed
@SCJMayAllocate({javax.safetycritical.annotate.AllocationContext.CURRENT})
@SCJMaySelfSuspend(false)
@SCJPhase({
    javax.safetycritical.annotate.Phase.STARTUP,
    javax.safetycritical.annotate.Phase.INITIALIZATION,
    javax.safetycritical.annotate.Phase.RUN,
    javax.safetycritical.annotate.Phase.CLEANUP })
public javax.realtime.device.RawShortReader createRawShortReader(
  RawMemoryRegion region,
  long base,
  int count,
  int stride)
  throws java.lang.SecurityException, javax.realtime.OffsetOutOfBoundsException,
       javax.realtime.SizeOutOfBoundsException,
      javax.realtime.MemoryTypeConflictException,
      javax.realtime.UnsupportedRawMemoryRegionException
```

Create an instance of a class that implements **javax.realtime.device.RawShortReader** and accesses memory of region in the address range described by base, stride, and count. The actual extent of the memory addressed by the object is stride x *size of RawShortReader* x count. The object is allocated in the current memory area of the calling thread.

region — The address space from which the new instance should be taken.

base — The starting physical address accessible through the returned instance.

count — The number of memory elements accessible through the returned instance.

stride — The distance to the next element as a mulitple of element size, where 1 means the elements are adjacent in memory.

returns an object that implements **javax.realtime.device.RawShortReader** and supports access to the specified range in the memory region.

Throws IllegalArgumentException when base is negative, count is not greater than zero, or stride is less than one.

Throws SecurityException when the caller does not have permissions to access the given memory region or the specified range of addresses.

Throws OffsetOutOfBoundsException when base is invalid.

Throws SizeOutOfBoundsException when the memory addressed by the object would extend into an invalid range of memory.

Throws MemoryTypeConflictException when base does not point to a memory that matches the type served by this factory.

```
@SCJAllowed
@SCJMayAllocate({javax.safetycritical.annotate.AllocationContext.CURRENT})
@SCJMaySelfSuspend(false)
@SCJPhase({
    javax.safetycritical.annotate.Phase.STARTUP,
    javax.safetycritical.annotate.Phase.INITIALIZATION,
    javax.safetycritical.annotate.Phase.RUN,
    javax.safetycritical.annotate.Phase.CLEANUP })
public javax.realtime.device.RawShortWriter createRawShortWriter(
  RawMemoryRegion region,
  long base,
  int count,
  int stride)
  throws java.lang.SecurityException, javax.realtime.OffsetOutOfBoundsException,
       javax.realtime.SizeOutOfBoundsException,
      javax.realtime.MemoryTypeConflictException,
      javax.realtime.UnsupportedRawMemoryRegionException
```

Create an instance of a class that implements **javax.realtime.device.RawShortWriter** and accesses memory of region in the address range described by base, stride, and count. The actual extent of the memory addressed by the object is stride x *size of RawShortWriter* x count. The object is allocated in the current memory area of the calling thread.

region — The address space from which the new instance should be taken.

base — The starting physical address accessible through the returned instance.

count — The number of memory elements accessible through the returned instance.

stride — The distance to the next element as a mulitple of element size, where 1 means the elements are adjacent in memory.

returns an object that implements **javax.realtime.device.RawShortWriter** and supports access to the specified range in the memory region.

Throws IllegalArgumentException when base is negative, count is not greater than zero, or stride is less than one.

Throws SecurityException when the caller does not have permissions to access the given memory region or the specified range of addresses.

Throws OffsetOutOfBoundsException when base is invalid.

Throws SizeOutOfBoundsException when the memory addressed by the object would extend into an invalid range of memory.

Throws MemoryTypeConflictException when base does not point to a memory that matches the type served by this factory.

```
@SCJAllowed
@SCJMayAllocate({})
@SCJMaySelfSuspend(false)
@SCJPhase({
    javax.safetycritical.annotate.Phase.STARTUP,
    javax.safetycritical.annotate.Phase.INITIALIZATION,
    javax.safetycritical.annotate.Phase.RUN,
    javax.safetycritical.annotate.Phase.CLEANUP })
public void deregister(RawMemoryRegionFactory factory)
  throws javax.realtime.DeregistrationException
```

Remove support for a new memory region

factory — is the **javax.realtime.device.RawMemoryRegionFactory** to make unavailable.

Throws RegistrationException when the factory is not registered.

```
@SCJAllowed
@SCJMaySelfSuspend(false)
@SCJPhase({
    javax.safetycritical.annotate.Phase.STARTUP,
    javax.safetycritical.annotate.Phase.INITIALIZATION,
    javax.safetycritical.annotate.Phase.RUN,
    javax.safetycritical.annotate.Phase.CLEANUP})
@SCJMayAllocate({})
public static javax.realtime.device.RawMemoryFactory getDefaultFactory( )
```

Get the factory with support for the platform defined regions.

returns the platform defined factory

```
@SCJAllowed
@SCJMayAllocate({})
@SCJMaySelfSuspend(false)
@SCJPhase({
    javax.safetycritical.annotate.Phase.STARTUP,
    javax.safetycritical.annotate.Phase.INITIALIZATION,
    javax.safetycritical.annotate.Phase.RUN,
    javax.safetycritical.annotate.Phase.CLEANUP })
public void register(RawMemoryRegionFactory factory)
  throws javax.realtime.RegistrationException
```

Add support for a new memory region

factory — is the **javax.realtime.device.RawMemoryRegionFactory** to use for creating **javax.realtime.device.RawMemory** objects for the **javax.realtime.device-.RawMemoryRegion** s it makes available.

Throws RegistrationException when the factory already is already registered.

E.3.3 CLASS **RawMemoryRegion**

@SCJAllowed
public class RawMemoryRegion **extends** java.lang.Object

RawMemoryRegion is a class for typing raw memory regions. It is returned by the RawMemoryRegionFactory.getRegion methods of the raw memory region factory classes, and it is used with methods such as RawMemoryFactory.createRawByte(RawMemoryRegion, long, int, int) and RawMemoryFactory.createRawDouble(RawMemoryRegion, long, int, int} methods to identify the region from which the application wants to get an accessor instance.

Constructors

```
@SCJAllowed
@SCJMayAllocate({})
@SCJMaySelfSuspend(false)
@SCJPhase({
    javax.safetycritical.annotate.Phase.STARTUP,
    javax.safetycritical.annotate.Phase.INITIALIZATION,
    javax.safetycritical.annotate.Phase.RUN,
    javax.safetycritical.annotate.Phase.CLEANUP})
public RawMemoryRegion(String name)
```

Methods

```
@SCJAllowed
@SCJMayAllocate({})
@SCJMaySelfSuspend(false)
@SCJPhase({
    javax.safetycritical.annotate.Phase.STARTUP,
    javax.safetycritical.annotate.Phase.INITIALIZATION,
    javax.safetycritical.annotate.Phase.RUN,
    javax.safetycritical.annotate.Phase.CLEANUP})
public final java.lang.String getName( )
```

Obtains the name of this region type.

returns the region types name

```
@SCJAllowed
@SCJMaySelfSuspend(false)
@SCJPhase({
    javax.safetycritical.annotate.Phase.STARTUP,
    javax.safetycritical.annotate.Phase.INITIALIZATION,
    javax.safetycritical.annotate.Phase.RUN,
```

```
    javax.safetycritical.annotate.Phase.CLEANUP})
@SCJMayAllocate({
    javax.safetycritical.annotate.AllocationContext.CURRENT,
    javax.safetycritical.annotate.AllocationContext.INNER,
    javax.safetycritical.annotate.AllocationContext.OUTER})
public static javax.realtime.device.RawMemoryRegion getRegion(String name)
```

Get a region type when it already exists or creates a new one.

name — of the region

returns the region type object.

```
@SCJAllowed
@SCJMayAllocate({})
@SCJMaySelfSuspend(false)
@SCJPhase({
    javax.safetycritical.annotate.Phase.STARTUP,
    javax.safetycritical.annotate.Phase.INITIALIZATION,
    javax.safetycritical.annotate.Phase.RUN,
    javax.safetycritical.annotate.Phase.CLEANUP})
public static boolean isRawMemoryRegion(String name)
```

Ask whether or not there is a memory region type of a given name.

name — for which to search

returns true when there is one and false otherwise.

```
@SCJAllowed
@SCJMayAllocate({})
@SCJMaySelfSuspend(false)
@SCJPhase({
    javax.safetycritical.annotate.Phase.STARTUP,
    javax.safetycritical.annotate.Phase.INITIALIZATION,
    javax.safetycritical.annotate.Phase.RUN,
    javax.safetycritical.annotate.Phase.CLEANUP})
public final java.lang.String toString( )
```

Gets a printable representation for a Region.

returns the name of this memory region type.

Appendix F

Javadoc Description of Package javax.realtime.memory

Package Contents *Page*

Classes

F.1 Classes

F.2 Interfaces

F.3 Classes

F.3.1 CLASS ScopeParameters

```
@SCJAllowed
public class ScopeParameters extends javax.realtime.MemoryParameters
```

Extend memory parameters to provide limits for scoped memory.

See Also: javax.realtime.MemoryParameters

Constructors

```
@SCJAllowed
@SCJMayAllocate({
    javax.safetycritical.annotate.AllocationContext.CURRENT,
    javax.safetycritical.annotate.AllocationContext.INNER,
    javax.safetycritical.annotate.AllocationContext.OUTER})
@SCJMaySelfSuspend(true)
@SCJPhase({
    javax.safetycritical.annotate.Phase.STARTUP,
    javax.safetycritical.annotate.Phase.INITIALIZATION,
    javax.safetycritical.annotate.Phase.RUN,
    javax.safetycritical.annotate.Phase.CLEANUP })
public ScopeParameters(long maxInitialArea,
  long maxImmortal,
  long maxContainingArea,
  long maxInitialBackingStore)
  throws java.lang.IllegalArgumentException
```

Create a ScopeParameters instance with the given values.

maxInitialArea — a limit on the amount of memory the schedulable may allocate in its initial scoped memory area. Units are in bytes. When zero, no allocation is allowed in the memory area. When the initial memory area is not a ScopedMemory , this parameter has no effect. To specify no limit, use MemoryParameters.UNLIMITED .

maxImmortal — A limit on the amount of memory the schedulable may allocate in the immortal area. Units are in bytes. When zero, no allocation allowed in immortal. To specify no limit, use MemoryParameters.UNLIMITED .

maxContainingArea — a limit on the amount of memory the schedulable may allocate in memory area where it was created. Units are in bytes. When zero, no allocation is allowed in the memory area. When the containing memory area is not a ScopedMemory , this parameter has no effect. To specify no limit, use MemoryParameters.UNLIMITED . For schedulables created within a mission, the containing memory area is Mission memory. For the initial MissionSequencer, the initial memory area is Immortal memory.

maxInitialBackingStore — A limit on the amount of backing store this task may allocate from backing store of its inital area, when that is a stacked memory. Units are in bytes. When zero, no allocation is allowed in that memory area. Backing store that is returned to the global backing store is subtracted from the limit. To specify no limit, use MemoryParameters.UNLIMITED .

Throws IllegalArgumentException when any value other than positive, zero, or **javax.realtime.MemoryParametersUNREFERENCED** is passed as the value of maxInitialArea , maxImmortal , maxParentBackingStore , or maxContainingArea.

```
@SCJAllowed
@SCJMayAllocate({
   javax.safetycritical.annotate.AllocationContext.CURRENT,
   javax.safetycritical.annotate.AllocationContext.INNER,
   javax.safetycritical.annotate.AllocationContext.OUTER})
@SCJMaySelfSuspend(true)
@SCJPhase({
   javax.safetycritical.annotate.Phase.STARTUP,
   javax.safetycritical.annotate.Phase.INITIALIZATION,
   javax.safetycritical.annotate.Phase.RUN,
   javax.safetycritical.annotate.Phase.CLEANUP })
public ScopeParameters(long maxInitialArea,
  long maxImmortal,
  long maxInitialBackingStore)
  throws java.lang.IllegalArgumentException
```

Same as ScopeParameters(maxInitialArea, maxImmortal, maxParentBackingStore, MemoryParameters.UNLIMITED).

Methods

```
@SCJAllowed
@SCJMaySelfSuspend(false)
@SCJPhase({
   javax.safetycritical.annotate.Phase.INITIALIZATION,
   javax.safetycritical.annotate.Phase.RUN,
   javax.safetycritical.annotate.Phase.CLEANUP})
@SCJMayAllocate({})
public long getMaxBackingStore( )
```

Determine the limit on backing store for this task.

returns the limit on backing store.

```
@SCJMaySelfSuspend(false)
@SCJPhase({
    javax.safetycritical.annotate.Phase.INITIALIZATION,
    javax.safetycritical.annotate.Phase.RUN,
    javax.safetycritical.annotate.Phase.CLEANUP})
@SCJMayAllocate({})
@SCJAllowed
public long getMaxContainingArea( )
```

Determine the limit on allocation in the area where the task was created.

returns the limit on allocation in the area where the task was created.

F.3.2 CLASS ScopedCycleException

```
@SCJAllowed
public class ScopedCycleException implements java.io.Serializable extends
      java.lang.RuntimeException
```

Constructors

```
@SCJMayAllocate({javax.safetycritical.annotate.AllocationContext.CURRENT})
@SCJAllowed
@SCJMaySelfSuspend(false)
@SCJPhase({
    javax.safetycritical.annotate.Phase.STARTUP,
    javax.safetycritical.annotate.Phase.INITIALIZATION,
    javax.safetycritical.annotate.Phase.RUN,
    javax.safetycritical.annotate.Phase.CLEANUP })
public ScopedCycleException( )
```

```
@SCJMayAllocate({javax.safetycritical.annotate.AllocationContext.CURRENT})
@SCJAllowed
@SCJMaySelfSuspend(false)
@SCJPhase({
    javax.safetycritical.annotate.Phase.STARTUP,
    javax.safetycritical.annotate.Phase.INITIALIZATION,
    javax.safetycritical.annotate.Phase.RUN,
    javax.safetycritical.annotate.Phase.CLEANUP })
public ScopedCycleException(String description)
```

F.3.3 CLASS **ScopedMemory**

```
@SCJAllowed
public abstract class ScopedMemory extends javax.realtime.MemoryArea
```

Scoped memory implements the scoped allocation context.

Methods

```
@SCJAllowed
@SCJMaySelfSuspend(false)
@SCJPhase({
    javax.safetycritical.annotate.Phase.INITIALIZATION,
    javax.safetycritical.annotate.Phase.RUN,
    javax.safetycritical.annotate.Phase.CLEANUP})
@SCJMayAllocate({})
public long backingStoreConsumed( )
```

Determines the amount of backing store consumed by this scoped memory and its children.

returns the total amount of backing store.

```
@SCJAllowed
@SCJMaySelfSuspend(false)
@SCJPhase({
    javax.safetycritical.annotate.Phase.INITIALIZATION,
    javax.safetycritical.annotate.Phase.RUN,
    javax.safetycritical.annotate.Phase.CLEANUP})
@SCJMayAllocate({})
public long backingStoreRemaining( )
```

Determines the remaining amount of backing store available to this scoped memory and its children.

returns the total amount of backing store.

```
@SCJAllowed
@SCJMaySelfSuspend(false)
@SCJPhase({
    javax.safetycritical.annotate.Phase.INITIALIZATION,
    javax.safetycritical.annotate.Phase.RUN,
    javax.safetycritical.annotate.Phase.CLEANUP})
@SCJMayAllocate({})
public long backingStoreSize( )
```

Determines the total amount of backing store for this scoped memory and its children.

returns the total amount of backing store.

F.3.4 CLASS **StackedMemory**

@SCJAllowed
public class StackedMemory **extends** javax.realtime.memory.ScopedMemory

This class can not be instantiated in SCJ. It is subclassed by MissionMemory and PrivateMemory. It has no visible methods for SCJ applications.

Appendix G

Javadoc Description of Package javax.safetycritical

SCJ provides some additional classes to provide the mission framework and handle startup and shutdown of safety-critical applications.*Package Contents*

Interfaces

Classes

G.1 Classes

G.2 Interfaces

G.2.1 INTERFACE ManagedSchedulable

```
@SCJAllowed
public interface ManagedSchedulable extends javax.realtime.BoundSchedulable
```

In SCJ, all schedulable objects are managed by a mission.

This interface is implemented by all SCJ Schedulable classes. It defines the mechanism by which the ManagedSchedulable is registered with the mission for its management. This interface is used by SCJ classes. It is not intended for direct use by application classes.

Methods

```
@SCJAllowed(javax.safetycritical.annotate.Level.SUPPORT)
@SCJPhase({javax.safetycritical.annotate.Phase.CLEANUP})
@SCJMaySelfSuspend(false)
@SCJMayAllocate({
   javax.safetycritical.annotate.AllocationContext.CURRENT,
   javax.safetycritical.annotate.AllocationContext.INNER,
   javax.safetycritical.annotate.AllocationContext.OUTER})
public void cleanUp( )
```

Runs any end-of-mission clean up code associated with this schedulable object.

Application developers implement this method with code to be executed when this schedulable object's execution is disabled (after termination has been requested of the enclosing mission).

When the cleanUp method is called, the private memory area associated with this schedulable object shall be the current memory area. If desired, the cleanUp method may introduce new private memory areas. The memory allocated to ManagedSchedulables shall be available to be reclaimed when its Mission's cleanUp method returns.

```
@SCJAllowed
@SCJPhase({javax.safetycritical.annotate.Phase.INITIALIZATION})
@SCJMayAllocate({})
@SCJMaySelfSuspend(false)
public void register( )
```

Register this managed schedulable with the current mission.

```
@SCJAllowed(javax.safetycritical.annotate.Level.SUPPORT)
@SCJMayAllocate({})
@SCJMaySelfSuspend(false)
@SCJPhase({
   javax.safetycritical.annotate.Phase.RUN,
   javax.safetycritical.annotate.Phase.CLEANUP})
public void signalTermination( )
```

Called by the infrastructure to indicate that the enclosing mission has been instructed to terminate.

The application can override the default implementations of signalTermination() to facilitate termination of the ManagedSchedulable.

G.2.2 INTERFACE **Safelet**

```
@SCJAllowed
public interface Safelet
```

A safety-critical application consists of one or more missions, executed concurrently or in sequence. Every safety-critical application must implement Safelet which identifies the outer-most MissionSequencer. This outer-most MissionSequencer runs the sequence of missions that comprise this safety-critical application.

The mechanism used to identify the Safelet to a particular SCJ environment is implementation defined.

Fields

```
@SCJAllowed
public static final long INSUFFICIENT_BACKING_STORE
```

```
@SCJAllowed
public static final long INSUFFICIENT_IMMORTAL_MEMORY
```

Methods

```
@SCJAllowed(javax.safetycritical.annotate.Level.SUPPORT)
@SCJPhase({javax.safetycritical.annotate.Phase.CLEANUP})
@SCJMayAllocate({javax.safetycritical.annotate.AllocationContext.CURRENT})
@SCJMaySelfSuspend(false)
public void cleanUp( )
```

Called by the infrastructure after termination of the MissionSequencer for this Safelet.

```
@SCJAllowed(javax.safetycritical.annotate.Level.SUPPORT)
@SCJPhase({javax.safetycritical.annotate.Phase.STARTUP})
@SCJMayAllocate({javax.safetycritical.annotate.AllocationContext.CURRENT})
@SCJMaySelfSuspend(false)
public javax.safetycritical.MissionSequencer getSequencer( )
```

The infrastructure invokes getSequencer to obtain the MissionSequencer object that oversees execution of missions for this application. The returned MissionSequencer resides in immortal memory.

returns the MissionSequencer that oversees execution of missions for this application.

```
@SCJAllowed(javax.safetycritical.annotate.Level.SUPPORT)
@SCJPhase({javax.safetycritical.annotate.Phase.STARTUP})
@SCJMayAllocate({})
@SCJMaySelfSuspend(false)
public long globalBackingStoreSize( )
```

returns the amount of additional backing store memory that must be available for managed memory areas. If the amount of remaining memory is less than this requested size, the infrastructure shall call the handleStartupError() method to determine whether the application should be immediately halted.

```
@SCJAllowed(javax.safetycritical.annotate.Level.SUPPORT)
@SCJPhase({javax.safetycritical.annotate.Phase.STARTUP})
@SCJMayAllocate({javax.safetycritical.annotate.AllocationContext.CURRENT})
@SCJMaySelfSuspend(false)
public boolean handleStartupError(int cause, long val)
```

Called during startup by the infrastructure if the infrastructure detects the presence of a fatal startup error allocating memory or for any other implementation defined reason. This method returns a boolean indication whether it intends for the infrastructure to immediately halt execution, or whether it intends for the infrastructure to retry the failed allocation request. This method makes it possible for an application to attempt to execute in a degraded mode in the event of certain types of failures, such as a partial memory failure.

returns True if the infrastructure should immediately halt as a result of detecting the fatal startup error. If False is returned, the infrastructure should repeat its calls to immortalMemorySize() and globalBackingStoreSize(), providing the application the ability to reconfigure itself, if possible, to work around the fatal startup error.

cause — Identifies the condition that caused the infrastructure to call this method. If cause = INSUFFICIENT_IMMORTAL_MEMORY, the amount of available memory is insufficient for the immortal memory requested by the previous call to immortalMemorySize(). If cause = INSUFFICIENT_BACKING_STORE, the amount of

available memory is insufficient for the backing store memory requested by the previous call to globalBackingStoreSize(). If cause has any other value, its meaning is implementation defined.

val — If cause = INSUFFICIENT_IMMORTAL_MEMORY, val contains the shortfall in available memory for the immortal memory requested by the previous call to immortalMemorySize(). If cause = INSUFFICIENT_BACKING_STORE, val contains the shortfall in available memory for the backing store memory requested by the previous call to globalBackingStoreSize(). If cause has any other value, the meaning of val is implementation defined.

```
@SCJAllowed(javax.safetycritical.annotate.Level.SUPPORT)
@SCJPhase({javax.safetycritical.annotate.Phase.STARTUP})
@SCJMayAllocate({})
@SCJMaySelfSuspend(false)
public long immortalMemorySize( )
```

returns the amount of additional immortal memory that must be available for allocations to be performed by this application. If the amount of remaining memory is less than this requested size, the infrastructure shall call the handleStartupError() method to determine whether the application should be immediately halted.

```
@SCJAllowed(javax.safetycritical.annotate.Level.SUPPORT)
@SCJPhase({javax.safetycritical.annotate.Phase.STARTUP})
@SCJMayAllocate({javax.safetycritical.annotate.AllocationContext.CURRENT})
@SCJMaySelfSuspend(true)
public void initializeApplication( )
```

The infrastructure shall invoke initializeApplication in the allocation context of immortal memory. The application can use this method to allocate data structures in immortal memory. This method shall be called exactly once by the infrastructure.

G.3 Classes

G.3.1 CLASS AperiodicEventHandler

```
@SCJAllowed(javax.safetycritical.annotate.Level.LEVEL_1)
public abstract class AperiodicEventHandler extends
    javax.safetycritical.ManagedEventHandler
```

This class encapsulates an aperiodic event handler. Concrete subclasses must implement the handleAsyncEvent method and may override the default cleanUp method.

There is no application access to the RTSJ fireCount mechanisms, so the associated methods are missing; see the AperiodicParameters class description for

additional information.

Note that the values in parameters passed to the constructors are those that will be used by the infrastructure. Changing these values after construction will have no impact on the created event handler.

Constructors

```
@SCJAllowed(javax.safetycritical.annotate.Level.LEVEL_1)
@SCJPhase({javax.safetycritical.annotate.Phase.INITIALIZATION})
@SCJMaySelfSuspend(false)
@SCJMayAllocate({
   javax.safetycritical.annotate.AllocationContext.CURRENT,
   javax.safetycritical.annotate.AllocationContext.INNER,
   javax.safetycritical.annotate.AllocationContext.OUTER})
public AperiodicEventHandler(PriorityParameters priority,
  AperiodicParameters release,
  ScopeParameters storage,
  ConfigurationParameters config)
```

Constructs an aperiodic event handler that can be explicitly released.

The values passed as constructor parameters are captured at construction time. Any subsequent mutation of the parameter objects has no effect on the behavior of the constructed object.

priority — specifies the priority parameters for this handler. Must not be null.
release — specifies the release parameters for this handler. A null parameter indicates that there is no deadline associated with this handler.
storage — - it must not be null. specifies the ScopeParameters for this handler.
config — specifies the ConfigurationParameters for this handler.
Throws IllegalArgumentException if priority or storage is null; or when any deadline miss handler specified is not an AperiodicEventHandler.

Methods

```
@SCJAllowed
@SCJMaySelfSuspend(false)
@SCJMayAllocate({})
@SCJPhase({
   javax.safetycritical.annotate.Phase.STARTUP,
   javax.safetycritical.annotate.Phase.INITIALIZATION,
   javax.safetycritical.annotate.Phase.RUN,
   javax.safetycritical.annotate.Phase.CLEANUP })
public final void release( )
```

Release this aperiodic event handler.

G.3.2 CLASS AperiodicLongEventHandler

```
@SCJAllowed(javax.safetycritical.annotate.Level.LEVEL_1)
public abstract class AperiodicLongEventHandler extends
    javax.safetycritical.ManagedLongEventHandler
```

This class encapsulates an aperiodic event handler that is passed a long value when it is released. Concrete subclasses must implement the handleAsyncEvent method and may override the default cleanUp method.

There is no programmer access to the RTSJ fireCount mechanisms, so the associated methods are missing; see the AperiodicParameters class description for additional information.

Note that the values in parameters classes passed to the constructors are those that will be used by the infrastructure. Changing these values after construction will have no impact on the created event handler.

Constructors

```
@SCJAllowed(javax.safetycritical.annotate.Level.LEVEL_1)
@SCJPhase({javax.safetycritical.annotate.Phase.INITIALIZATION})
@SCJMaySelfSuspend(false)
@SCJMayAllocate({
   javax.safetycritical.annotate.AllocationContext.CURRENT,
   javax.safetycritical.annotate.AllocationContext.INNER,
   javax.safetycritical.annotate.AllocationContext.OUTER})
public AperiodicLongEventHandler(PriorityParameters priority,
  AperiodicParameters release,
  ScopeParameters storage,
  ConfigurationParameters config)
```

Constructs an aperiodic long event handler that can be released.

The values passed as constructor parameters are captured at construction time. Any subsequent mutation of the parameter objects has no effect on the behavior of the constructed object.

priority — specifies the priority parameters for this handler; it must not be null.
release — specifies the release parameters for this handler. A null parameter indicates that there is no deadline associated with this handler.
storage — specifies the ScopeParameters for this handler; it must not be null.
config — specifies the ConfigurationParameters for this handler.
Throws IllegalArgumentException if priority or storge is null; or when any deadline miss handler specified in release is not an AperiodicHandler.

Methods

```
@SCJAllowed
```

```
@SCJMaySelfSuspend(false)
@SCJMayAllocate({})
@SCJPhase({
    javax.safetycritical.annotate.Phase.STARTUP,
    javax.safetycritical.annotate.Phase.INITIALIZATION,
    javax.safetycritical.annotate.Phase.RUN,
    javax.safetycritical.annotate.Phase.CLEANUP })
public final void release(long data)
```

Release this long aperiodic event handler.

data — is the value associated with the release.

G.3.3 CLASS CyclicExecutive

```
@SCJAllowed
public abstract class CyclicExecutive extends javax.safetycritical.Mission
```

A CyclicExecutive represents a Level 0 mission. Every mission in a Level 0 application must be a subclass of CyclicExecutive.

Constructors

```
@SCJAllowed
@SCJPhase({
    javax.safetycritical.annotate.Phase.STARTUP,
    javax.safetycritical.annotate.Phase.INITIALIZATION})
@SCJMaySelfSuspend(false)
@SCJMayAllocate({
    javax.safetycritical.annotate.AllocationContext.CURRENT,
    javax.safetycritical.annotate.AllocationContext.INNER,
    javax.safetycritical.annotate.AllocationContext.OUTER})
public CyclicExecutive( )
```

Construct a CyclicExecutive object.

Methods

```
@SCJAllowed(javax.safetycritical.annotate.Level.SUPPORT)
@SCJPhase({
    javax.safetycritical.annotate.Phase.STARTUP,
    javax.safetycritical.annotate.Phase.INITIALIZATION})
@SCJMaySelfSuspend(false)
@SCJMayAllocate({
    javax.safetycritical.annotate.AllocationContext.CURRENT,
    javax.safetycritical.annotate.AllocationContext.INNER,
    javax.safetycritical.annotate.AllocationContext.OUTER})
public abstract javax.safetycritical.CyclicSchedule getSchedule(
  PeriodicEventHandler [] handlers)
```

Every CyclicExecutive shall provide its own cyclic schedule, which is represented by an instance of the CyclicSchedule class. Application programmers are expected to implement this method to provide a schedule that is appropriate for the mission.

Level 0 infrastructure code invokes the getSchedule method on the mission returned from MissionSequencer.getNextMission after invoking the mission's initialize method in order to obtain the desired cyclic schedule. Upon entry into the getSchedule method, this mission's mission memory area shall be the active allocation context. The value returned from getSchedule shall reside in the current mission's mission memory area area or in some enclosing scope.

Infrastructure code shall check that all of the PeriodicEventHandler objects referenced from within the returned CyclicSchedule object have been registered for execution with this Mission. If not, the infrastructure shall immediately terminate execution of this mission without executing any event handlers.

handlers — represents all of the handlers that have been registered with this Mission. The entries in the handlers array are sorted in the same order in which they were registered by the corresponding CyclicExecutive's initialize method. The infrastructure shall copy the information in the handlers array into its private memory, so subsequent application changes to the handlers array will have no effect.

returns the schedule to be used by the CyclicExecutive.

G.3.4 CLASS CyclicSchedule

```
@SCJAllowed
public final class CyclicSchedule extends java.lang.Object
```

A CyclicSchedule object represents a time-driven sequence of firings for deterministic scheduling of periodic event handlers. The static cyclic scheduler repeatedly executes the firing sequence.

Constructors

```
@SCJAllowed
@SCJPhase({javax.safetycritical.annotate.Phase.INITIALIZATION})
@SCJMaySelfSuspend(false)
@SCJMayAllocate({
   javax.safetycritical.annotate.AllocationContext.CURRENT,
   javax.safetycritical.annotate.AllocationContext.INNER,
   javax.safetycritical.annotate.AllocationContext.OUTER})
public CyclicSchedule(CyclicSchedule.Frame [] frames)
  throws java.lang.IllegalArgumentException,
     java.lang.IllegalStateException
```

Construct a cyclic schedule by copying the frames array into a private array within the same memory area as this newly constructed CyclicSchedule object.

The frames array represents the order in which event handlers are to be scheduled. Note that some Frame entries within this array may have zero PeriodicEventHandlers associated with them. This would represent a period of time during which the CyclicExecutive is idle.

Throws IllegalArgumentException if any element of the frames array equals null or if the frames array is empty,
Throws IllegalStateException if invoked by a Level 1 a Level 2 application.

G.3.5 CLASS CyclicSchedule.Frame

```
@SCJAllowed
public static final class CyclicSchedule.Frame extends java.lang.Object
```

A time slot within the cycle.

Constructors

```
@SCJAllowed
@SCJMaySelfSuspend(false)
@SCJMayAllocate({
   javax.safetycritical.annotate.AllocationContext.CURRENT,
   javax.safetycritical.annotate.AllocationContext.INNER,
   javax.safetycritical.annotate.AllocationContext.OUTER})
@SCJPhase({
   javax.safetycritical.annotate.Phase.STARTUP,
   javax.safetycritical.annotate.Phase.INITIALIZATION,
   javax.safetycritical.annotate.Phase.RUN,
   javax.safetycritical.annotate.Phase.CLEANUP })
public CyclicSchedule.Frame(RelativeTime duration, PeriodicEventHandler [] handlers)
```

Create a scheduling frame with a duration of execution and a set of handlers that are to be execute in order when the frame is run. It allocates private copies of the array that holds the set of handlers in the same memory area as the object itself. This ensures that the array cannot be changed by the application. One should note that even though handlers is declared as enclosing this object only its contents must actually enclose this object.

duration — is the time the frame has to execute
handlers — is the set of handlers that are run in the frame

G.3.6 CLASS Frame

```
@SCJAllowed
public final class Frame extends java.lang.Object
```

Constructors

```
@SCJAllowed
@SCJPhase({javax.safetycritical.annotate.Phase.INITIALIZATION})
@SCJMaySelfSuspend(false)
@SCJMayAllocate({
   javax.safetycritical.annotate.AllocationContext.CURRENT,
   javax.safetycritical.annotate.AllocationContext.INNER,
   javax.safetycritical.annotate.AllocationContext.OUTER})
public Frame(RelativeTime duration, PeriodicEventHandler [] handlers)
```

Allocates and retains private shallow copies of the duration and handlers array within the same memory area as this. The elements within the copy of the handlers array are the exact same elements as in the handlers array. Thus, it is essential that the elements of the handlers array reside in memory areas that enclose this. Usually, this Frame object is instantiated within the mission memory area that corresponds to the Level 0 mission that is to be scheduled.

Within each execution frame of the CyclicSchedule, the PeriodicEventHandler objects represented by the handlers array will be released in the same order as they appear within this array.

G.3.7 CLASS LinearMissionSequencer

```
@SCJAllowed
public class LinearMissionSequencer extends
    javax.safetycritical.MissionSequencer
```

A LinearMissionSequencer is a MissionSequencer that serves the needs of a common design pattern in which the sequence of Mission executions is known prior to execution and all missions can be preallocated within an outer-nested memory area.

The parameter <M> allows application code to differentiate between LinearMissionSequencers that are designed for use in Level 0 vs. other environments. For example, a LinearMissionSequencer<CyclicExecutive> is known to only run missions that are suitable for execution within a Level 0 run-time environment.

Constructors

```
@SCJAllowed
@SCJPhase({javax.safetycritical.annotate.Phase.INITIALIZATION})
@SCJMaySelfSuspend(false)
@SCJMayAllocate({
```

```
    javax.safetycritical.annotate.AllocationContext.CURRENT,
    javax.safetycritical.annotate.AllocationContext.INNER,
    javax.safetycritical.annotate.AllocationContext.OUTER})
public LinearMissionSequencer(PriorityParameters priority,
  ScopeParameters storage,
  ConfigurationParameters config,
  Mission mission,
  boolean repeat,
  String name)
  throws java.lang.IllegalArgumentException,
      java.lang.IllegalStateException
```

Construct a LinearMissionSequencer object to oversee execution of a single mission m.

priority — The priority at which the MissionSequencer's bound thread executes.

storage — The memory resources to be dedicated to execution of this MissionSequencer's bound thread.

config — The configuration parameters to be dedicated to execution of this MissionSequencer's bound thread.

mission — The single mission that runs under the oversight of this LinearMissionSequencer.

repeat — When repeat is true, the specified mission shall be repeated indefinitely.

name — The name by which this LinearMissionSequencer will be identified in traces for use in debug or in toString.

Throws IllegalArgumentException if any of the arguments equals null.

```
@SCJAllowed
@SCJPhase({javax.safetycritical.annotate.Phase.INITIALIZATION})
@SCJMaySelfSuspend(false)
@SCJMayAllocate({
    javax.safetycritical.annotate.AllocationContext.CURRENT,
    javax.safetycritical.annotate.AllocationContext.INNER,
    javax.safetycritical.annotate.AllocationContext.OUTER})
public LinearMissionSequencer(PriorityParameters priority,
  ScopeParameters storage,
  ConfigurationParameters config,
  Mission mission,
  boolean repeat)
  throws java.lang.IllegalArgumentException,
      java.lang.IllegalStateException
```

This constructor behaves the same as calling LinearMissionSequencer(PriorityParameters, ConfigurationParameters, boolean, M, String) with the arguments (priority, storage, repeat, mission, null).

```
@SCJAllowed
```

```
@SCJPhase({javax.safetycritical.annotate.Phase.INITIALIZATION})
@SCJMaySelfSuspend(false)
@SCJMayAllocate({
    javax.safetycritical.annotate.AllocationContext.CURRENT,
    javax.safetycritical.annotate.AllocationContext.INNER,
    javax.safetycritical.annotate.AllocationContext.OUTER})
public LinearMissionSequencer(PriorityParameters priority,
  ScopeParameters storage,
  ConfigurationParameters config,
  Mission [] missions,
  boolean repeat,
  String name)
  throws java.lang.IllegalArgumentException,
     java.lang.IllegalStateException
```

Construct a LinearMissionSequencer object to oversee execution of the sequence of missions represented by the missions parameter. The LinearMissionSequencer runs the sequence of missions identified in its missions array exactly once, from low to high index position within the array. The constructor allocates a copy of its missions array argument within the current scope, so changes to .the missions array following construction will have no effect.

priority — The priority at which the MissionSequencer's bound thread executes.

storage — The memory resources to be dedicated to execution of this MissionSequencer's bound thread.

repeat — When repeat is true, the specified list of missions shall be repeated indefinitely.

missions — An array representing the sequence of missions to be executed under the oversight of this LinearMissionSequencer. Requires that the elements of the missions array reside in a scope that encloses the scope of this. The missions array itself may reside in a more inner-nested temporary scope.

name — The name by which this LinearMissionSequencer will be identified in traces for use in debug or in toString.

Throws IllegalArgumentException if any of the arguments equals null.

```
@SCJAllowed
@SCJPhase({javax.safetycritical.annotate.Phase.INITIALIZATION})
@SCJMaySelfSuspend(false)
@SCJMayAllocate({
    javax.safetycritical.annotate.AllocationContext.CURRENT,
    javax.safetycritical.annotate.AllocationContext.INNER,
    javax.safetycritical.annotate.AllocationContext.OUTER})
public LinearMissionSequencer(PriorityParameters priority,
  ScopeParameters storage,
  ConfigurationParameters config,
  Mission [] missions,
  boolean repeat)
```

throws java.lang.IllegalArgumentException,
java.lang.IllegalStateException

Same as LinearMissionSequencer(PriorityParameters, ConfigurationParameters, M[], boolean, String) with the arguments (priority, storage, missions, repeat, null).

Methods

@SCJAllowed(javax.safetycritical.annotate.Level.SUPPORT)
@SCJPhase({javax.safetycritical.annotate.Phase.INITIALIZATION})
@SCJMaySelfSuspend(false)
@SCJMayAllocate({})
@Override
protected final javax.safetycritical.Mission getNextMission( )

Returns a reference to the next Mission in the sequence of missions that was specified by the m or missions argument to this object's constructor.

See javax.safetycritical.MissionSequencer. getNextMission()

G.3.8 CLASS **ManagedEventHandler**

@SCJAllowed
public abstract class ManagedEventHandler **implements**
javax.safetycritical.ManagedSchedulable **extends**
javax.realtime.BoundAsyncEventHandler

In SCJ, all handlers must be registered with the enclosing mission, so SCJ applications use classes that are based on the ManagedEventHandler and the ManagedLongEventHandler class hierarchies. These class hierarchies allow a mission to manage all the handlers that are created during its initialization phase. The infrastructure sets up a private memory area for each managed handler that is entered before a call to handleAsyncEvent and is left on return.

The scheduling allocation domain of all managed event handlers is set, by default, to the scheduling allocation domain from where the associated mission initialization is being performed.

Methods

@SCJAllowed(javax.safetycritical.annotate.Level.SUPPORT)
@SCJPhase({javax.safetycritical.annotate.Phase.CLEANUP})
@SCJMaySelfSuspend(false)
@Override
@SCJMayAllocate({
javax.safetycritical.annotate.AllocationContext.CURRENT,
javax.safetycritical.annotate.AllocationContext.INNER,

javax.safetycritical.annotate.AllocationContext.OUTER})
public void cleanUp( )

Runs any end-of-mission clean up code associated with this schedulable object.
see ManagedSchedulable.cleanUp()

@SCJAllowed
@SCJMaySelfSuspend(false)
@SCJMayAllocate({})
@SCJPhase({
javax.safetycritical.annotate.Phase.STARTUP,
javax.safetycritical.annotate.Phase.INITIALIZATION,
javax.safetycritical.annotate.Phase.RUN,
javax.safetycritical.annotate.Phase.CLEANUP })
public java.lang.String getName( )

returns a string name of this event handler. The actual object returned shall be the same object that was passed to the event handler constructor.

@SCJPhase({javax.safetycritical.annotate.Phase.INITIALIZATION})
@SCJMaySelfSuspend(false)
@SCJMayAllocate({})
@Override
@SCJAllowed
public void register( )

Register this event handler with the current mission.

see javax.safetycritical.ManagedSchedulable.register()

@SCJAllowed(javax.safetycritical.annotate.Level.SUPPORT)
@SCJMaySelfSuspend(false)
@SCJMayAllocate({})
@SCJPhase({
javax.safetycritical.annotate.Phase.RUN,
javax.safetycritical.annotate.Phase.CLEANUP})
public void signalTermination( )

Called by the infrastructure to indicate that the enclosing mission has been instructed to terminate. The default behavior is no action.

G.3.9 CLASS ManagedInterruptServiceRoutine

@SCJAllowed(javax.safetycritical.annotate.Level.LEVEL_1)
public abstract class ManagedInterruptServiceRoutine **extends**
javax.realtime.device.InterruptServiceRoutine

Constructors

```
@SCJAllowed(javax.safetycritical.annotate.Level.LEVEL_1)
@SCJMaySelfSuspend(false)
@SCJMayAllocate({})
@SCJPhase({javax.safetycritical.annotate.Phase.INITIALIZATION})
public ManagedInterruptServiceRoutine(long sizes)
```

Creates an interrupt service routine

sizes — defines the memory space required by the handle method.

Methods

```
@SCJAllowed(javax.safetycritical.annotate.Level.LEVEL_1)
@SCJPhase({
   javax.safetycritical.annotate.Phase.INITIALIZATION,
   javax.safetycritical.annotate.Phase.RUN,
   javax.safetycritical.annotate.Phase.CLEANUP})
@SCJMaySelfSuspend(false)
@SCJMayAllocate({})
@Override
public javax.realtime.Affinity getAffinity( )
```

Determine the affinity set instance associated with {@code task}.

returns The associated affinity.

```
@SCJAllowed(javax.safetycritical.annotate.Level.LEVEL_1)
@Override
@SCJPhase({javax.safetycritical.annotate.Phase.INITIALIZATION})
@SCJMaySelfSuspend(false)
@SCJMayAllocate({})
public final void register(int interrupt)
  throws javax.realtime.RegistrationException
```

Equivalent to register(interrupt, prio) where prio is the highest InterruptCeilingPriority defined.

Throws IllegalStateException if this method is not part of a Mission which is currently being initialized

```
@SCJAllowed(javax.safetycritical.annotate.Level.LEVEL_1)
@SCJPhase({javax.safetycritical.annotate.Phase.INITIALIZATION})
@SCJMaySelfSuspend(false)
@SCJMayAllocate({})
public final void register(int interrupt, int ceiling)
  throws javax.realtime.RegistrationException
```

Registers the ISR for the given interrupt with the current mission and sets the ceiling priority of this. The filling of the associated interrupt vector is deferred until the end of the initialisation phase.

interrupt — is the implementation-dependent id for the interrupt.

ceiling — is the required ceiling priority.

Throws IllegalArgumentException if the ceiling is lower than the interrupt priority.

Throws RegistrationException if this object is not part of a Mission which is currently being initialized.

```
@SCJAllowed(javax.safetycritical.annotate.Level.LEVEL_1)
@SCJPhase({javax.safetycritical.annotate.Phase.INITIALIZATION})
@SCJMaySelfSuspend(false)
@SCJMayAllocate({})
@Override
public void setAffinity(Affinity set)
  throws java.lang.IllegalArgumentException,
    javax.realtime.ProcessorAffinityException, java.lang.NullPointerException
```

Set the processor affinity of a {@code task} to {@code set} with immediate effect.

set — is the processor affinity

Throws IllegalArgumentException when the intersection of {@code set} the affinity of any {@code ThreadGroup} instance containing {@code task} is empty.

Throws ProcessorAffinityException is thrown when the runtime fails to set the affinity for platform-specific reasons.

Throws NullPointerException when {@code set} is {@code null}.

```
@SCJAllowed(javax.safetycritical.annotate.Level.LEVEL_1)
@SCJPhase({javax.safetycritical.annotate.Phase.RUN})
@SCJMaySelfSuspend(false)
@SCJMayAllocate({})
public void unhandledException(Exception except)
```

Called by the infrastructure if an exception propagates outside of the handle method.

except — is the uncaught exception.

```
@SCJAllowed(javax.safetycritical.annotate.Level.LEVEL_1)
@SCJAllowed(javax.safetycritical.annotate.Level.SUPPORT)
@SCJPhase({javax.safetycritical.annotate.Phase.CLEANUP})
@SCJMaySelfSuspend(false)
@SCJMayAllocate({})
@Override
public final void unregister( )
```

Unregisters the ISR with the current mission.

G.3.10 CLASS ManagedLongEventHandler

```
@SCJAllowed
public abstract class ManagedLongEventHandler implements
    javax.safetycritical.ManagedSchedulable extends
    javax.realtime.BoundAsyncLongEventHandler
```

In SCJ, all handlers must be registered with the enclosing mission, so SCJ applications use classes that are based on the ManagedEventHandler and the ManagedLongEventHandler class hierarchies. These class hierarchies allow a mission to manage all the handlers that are created during its initialization phase. The infrastructure sets up a private memory area for each managed handler that is entered before a call to handleAsyncEvent and is left on return. This class differs from ManagedEventHandler in that when it is released, a long integer is provided for use by the released event handler(s).

The scheduling allocation domain of all managed long event handlers is set, by default, to the scheduling allocation domain from where the associated mission initialization is being performed.

Methods

```
@SCJAllowed(javax.safetycritical.annotate.Level.SUPPORT)
@SCJPhase({javax.safetycritical.annotate.Phase.CLEANUP})
@SCJMaySelfSuspend(false)
@Override
@SCJMayAllocate({
   javax.safetycritical.annotate.AllocationContext.CURRENT,
   javax.safetycritical.annotate.AllocationContext.INNER,
   javax.safetycritical.annotate.AllocationContext.OUTER})
public void cleanUp( )
```

Runs any end-of-mission clean up code associated with this schedulable object. see ManagedSchedulable.cleanUp()

```
@SCJAllowed
@SCJMaySelfSuspend(false)
@SCJMayAllocate({})
@SCJPhase({
   javax.safetycritical.annotate.Phase.STARTUP,
   javax.safetycritical.annotate.Phase.INITIALIZATION,
   javax.safetycritical.annotate.Phase.RUN,
   javax.safetycritical.annotate.Phase.CLEANUP })
public java.lang.String getName( )
```

returns a string name for this handler, including its priority and its level.

```
@SCJPhase({javax.safetycritical.annotate.Phase.INITIALIZATION})
@SCJMaySelfSuspend(false)
@SCJMayAllocate({})
@SCJAllowed
```

public void register( )

Register this event handler with the current mission.

See javax.safetycritical.ManagedSchedulable.register()

Throws IllegalStateException if this is an instance of MissionSequencer and the current execution environment does not support Level 2 capabilities.

```
@SCJAllowed(javax.safetycritical.annotate.Level.SUPPORT)
@SCJMaySelfSuspend(false)
@SCJMayAllocate({})
@SCJPhase({
   javax.safetycritical.annotate.Phase.RUN,
   javax.safetycritical.annotate.Phase.CLEANUP})
public void signalTermination( )
```

Called by the infrastructure to indicate that the enclosing mission has been instructed to terminate. The default behaviour is to perform no action.

G.3.11 CLASS ManagedMemory

```
@SCJAllowed
public abstract class ManagedMemory extends
    javax.realtime.memory.StackedMemory
```

This is the base class for all safety-critical Java memory areas. This class is used by the SCJ infrastructure to manage all SCJ memory areas. This class has no constructors, so it cannot be extended by an application.

Methods

```
@SCJAllowed
@SCJMayAllocate({javax.safetycritical.annotate.AllocationContext.CURRENT})
@SCJMaySelfSuspend(false)
@SCJPhase({
   javax.safetycritical.annotate.Phase.STARTUP,
   javax.safetycritical.annotate.Phase.INITIALIZATION,
   javax.safetycritical.annotate.Phase.RUN,
   javax.safetycritical.annotate.Phase.CLEANUP })
public long backingStoreRemaining( )
```

This method determines the available memory for new objects in the current ManagedMemory area.

returns the size in bytes of the remaining available memory to in the ManagedMemory area.

```
@SCJAllowed
@SCJMaySelfSuspend(false)
@SCJPhase({
   javax.safetycritical.annotate.Phase.INITIALIZATION,
   javax.safetycritical.annotate.Phase.RUN,
   javax.safetycritical.annotate.Phase.CLEANUP})
@SCJMayAllocate({javax.safetycritical.annotate.AllocationContext.CURRENT})
public static void enterPrivateMemory(long size, Runnable logic)
  throws java.lang.IllegalStateException
```

> Invoke the run method of logic with a fresh private memory area that is immediately nested within the current ManagedMemory area, sized to provide size bytes of allocatable memory as the current allocation area. Each instance of ManagedMemory maintains at most one inner-nested private memory area. In the case that enterPrivateMemory is invoked multiple times from within a particular ManagedMemory area without exiting that area, the first invocation instantiates the inner-nested private memory area and subsequent invocations resize and reuse the previously allocated private memory area. This is different from the case in which enterPrivateMemory is invoked from within a newly entered inner-nested PrivateMemory area. In this case, invocation of enterPrivateMemory results in creation and sizing of a new inner-nested private memory area.

size — is the number of allocatable memory bytes for the inner-nested private memory area.

logic — provides the run method that is to be executed within the inner-nested private memory area.

Throws IllegalStateException if the current allocation area is not the top-most (most recently entered) scope for the current schedulable object. (This would happen, for example, if the current schedulable object is in an outer-nested context as a result of having invoked, for example, executeInAreaOf).

Throws OutOfMemoryError if the currently running thread lacks sufficient backing store to have an inner-nested private memory area with size allocatable bytes; or if this is the first invocation of enterPrivateMemory from within the current allocation area and the current allocation area lacks sufficient backing store to allocate the inner-nested private memory area object.

```
@SCJAllowed
@SCJMayAllocate({})
@SCJPhase({
   javax.safetycritical.annotate.Phase.INITIALIZATION,
   javax.safetycritical.annotate.Phase.RUN,
   javax.safetycritical.annotate.Phase.CLEANUP})
@SCJMaySelfSuspend(false)
public static void executeInAreaOf(Object obj, Runnable logic)
```

Change the allocation context to the outer memory area where the object obj is allocated and invoke the run method of the logic Runnable.

obj — is the object allocated in the memory area that is entered.
logic — is the code to be executed in the entered memory area.

```
@SCJAllowed
@SCJMayAllocate({})
@SCJMaySelfSuspend(false)
@SCJPhase({
   javax.safetycritical.annotate.Phase.INITIALIZATION,
   javax.safetycritical.annotate.Phase.RUN,
   javax.safetycritical.annotate.Phase.CLEANUP})
public static void executeInOuterArea(Runnable logic)
```

Change the allocation context to the immediate outer memory area and invoke the run method of the Runnable.

logic — is the code to be executed in the entered memory area.
Throws IllegalStateException if the current memory area is ImmortalMemory.

G.3.12 CLASS ManagedThread

```
@SCJAllowed(javax.safetycritical.annotate.Level.LEVEL_2)
public abstract class ManagedThread implements
      javax.safetycritical.ManagedSchedulable extends javax.realtime.RealtimeThread
```

This class enables a mission to keep track of all the no-heap realtime threads that are created during the mission's initialization phase.

Note that the values in parameters classes passed to the constructors are those that will be used by the infrastructure. Changing these values after construction will have no impact on the created no-heap real-time thread. Managed threads have no release parameters.

Constructors

```
@SCJPhase({javax.safetycritical.annotate.Phase.INITIALIZATION})
@SCJMaySelfSuspend(false)
@SCJAllowed
@SCJMayAllocate({
   javax.safetycritical.annotate.AllocationContext.CURRENT,
   javax.safetycritical.annotate.AllocationContext.INNER,
   javax.safetycritical.annotate.AllocationContext.OUTER})
public ManagedThread(PriorityParameters priority,
  ReleaseParameters release,
  ScopeParameters scope,
  ConfigurationParameters storage,
  Runnable logic)
```

Creates a thread that is managed by the enclosing mission.

The priority represented by PriorityParameters is consulted only once, at construction time.

priority — specifies the priority parameters for this managed thread; it must not be null.

storage — specifies the memory parameters for this thread. May not be null.

logic — the code for this managed thread.

Throws IllegalArgumentException if priority or storage is null.

Methods

```
@SCJAllowed(javax.safetycritical.annotate.Level.SUPPORT)
@SCJPhase({javax.safetycritical.annotate.Phase.CLEANUP})
@SCJMaySelfSuspend(false)
@Override
@SCJMayAllocate({
   javax.safetycritical.annotate.AllocationContext.CURRENT,
   javax.safetycritical.annotate.AllocationContext.INNER,
   javax.safetycritical.annotate.AllocationContext.OUTER})
public void cleanUp( )
```

Runs any end-of-mission clean up code associated with this schedulable object.

see ManagedSchedulable.cleanUp()

```
@SCJPhase({javax.safetycritical.annotate.Phase.INITIALIZATION})
@SCJMaySelfSuspend(false)
@SCJMayAllocate({})
@Override
@SCJAllowed
public void register( )
```

Register this managed thread with the current mission.

see javax.safetycritical.ManagedSchedulable.register().

```
@SCJPhase({javax.safetycritical.annotate.Phase.INITIALIZATION})
@SCJMaySelfSuspend(false)
@SCJMayAllocate({})
@SCJAllowed(javax.safetycritical.annotate.Level.LEVEL_2)
public static void setDefaultUncaughtExceptionHandler(
  Thread.UncaughtExceptionHandler eh)
```

This method is used by the application to define an exception handler that will handle uncaught exceptions.

```
@SCJAllowed(javax.safetycritical.annotate.Level.LEVEL_2)
@SCJPhase({javax.safetycritical.annotate.Phase.INITIALIZATION})
@SCJMaySelfSuspend(false)
@SCJMayAllocate({})
public void setUncaughtExceptionHandler(Thread.UncaughtExceptionHandler eh)
```

Set the current uncaught exception handler.

eh — the UncaughtExceptionHandler to be set for this managed thread. The eh argument must reside in a scope that encloses the scope of this.

```
@SCJAllowed(javax.safetycritical.annotate.Level.SUPPORT)
@SCJMaySelfSuspend(false)
@SCJMayAllocate({})
@SCJPhase({
   javax.safetycritical.annotate.Phase.RUN,
   javax.safetycritical.annotate.Phase.CLEANUP})
public void signalTermination( )
```

Called by the infrastructure to indicate that the enclosing mission has been instructed to terminate. The default behaviour is no action.

G.3.13 CLASS Mission

```
@SCJAllowed
public abstract class Mission extends java.lang.Object
```

A Safety Critical Java application is comprised of one or more missions. Each mission is implemented as a subclass of this abstract Mission class. A mission is comprised of one or more ManagedSchedulable objects, conceptually running as independent threads of control, and the data that is shared between them.

Constructors

```
@SCJAllowed
@SCJPhase({
   javax.safetycritical.annotate.Phase.STARTUP,
   javax.safetycritical.annotate.Phase.INITIALIZATION,
   javax.safetycritical.annotate.Phase.RUN})
@SCJMaySelfSuspend(false)
@SCJMayAllocate({
   javax.safetycritical.annotate.AllocationContext.CURRENT,
   javax.safetycritical.annotate.AllocationContext.INNER,
   javax.safetycritical.annotate.AllocationContext.OUTER})
public Mission(AbsoluteTime start)
```

Allocate and initialize data structures associated with a Mission implementation.

The constructor may allocate additional infrastructure objects within the same MemoryArea that holds the implicit this argument.

The amount of data allocated in the same MemoryArea as this by the Mission constructor is implementation-defined. Application code will need to know the amount of this data to properly size the containing scope.

start — an absolute time value at which the Mission's ManagedSchedulable objects will be released for the first time after mission initialization has been completed unless they are delayed by their own start times. If start is null, or if start has already passed when the Mission's ManagedSchedulable objects become ready for release, they shall be released immediately.

```
@SCJAllowed
@SCJPhase({
   javax.safetycritical.annotate.Phase.STARTUP,
   javax.safetycritical.annotate.Phase.INITIALIZATION,
   javax.safetycritical.annotate.Phase.RUN})
@SCJMaySelfSuspend(false)
@SCJMayAllocate({
   javax.safetycritical.annotate.AllocationContext.CURRENT,
   javax.safetycritical.annotate.AllocationContext.INNER,
   javax.safetycritical.annotate.AllocationContext.OUTER})
public Mission( )
```

This constructor is equivalent to Mission(null).

Methods

```
@SCJAllowed(javax.safetycritical.annotate.Level.SUPPORT)
@SCJPhase({javax.safetycritical.annotate.Phase.CLEANUP})
@SCJMaySelfSuspend(true)
@SCJMayAllocate({
   javax.safetycritical.annotate.AllocationContext.CURRENT,
   javax.safetycritical.annotate.AllocationContext.INNER,
   javax.safetycritical.annotate.AllocationContext.OUTER})
protected boolean cleanUp( )
```

Cleans data structures and machine state upon termination of this Mission's run phase. Infrastructure running the controlling MissionSequencer invokes cleanUp after all ManagedSchedulables registered with this Mission have terminated, but before control leaves the corresponding mission memory area.

returns True to indicate that the mission sequencer shall continue with its sequence of missions, False to indicate that the mission sequence should be terminated and no further missions started. The default implementation of cleanUp returns True.

```
@SCJAllowed
```

```
@SCJMayAllocate({})
@SCJMaySelfSuspend(false)
@SCJPhase({
    javax.safetycritical.annotate.Phase.STARTUP,
    javax.safetycritical.annotate.Phase.INITIALIZATION,
    javax.safetycritical.annotate.Phase.RUN,
    javax.safetycritical.annotate.Phase.CLEANUP })
public static javax.safetycritical.Mission getMission( )
```

Obtain the current mission.

returns the instance of the Mission that is currently active.

If called during the initialization or clean up phase, getMission() returns the mission that is currently being initialized or cleaned up. If called during the run phase, getMission() returns the mission in which the currently executing ManagedSchedulable was registered. If called during the start up phase, getMission() returns null.

```
@SCJAllowed
@SCJPhase({
    javax.safetycritical.annotate.Phase.INITIALIZATION,
    javax.safetycritical.annotate.Phase.RUN,
    javax.safetycritical.annotate.Phase.CLEANUP})
@SCJMaySelfSuspend(false)
@SCJMayAllocate({})
public javax.safetycritical.MissionSequencer getSequencer( )
```

returns the MissionSequencer that is overseeing execution of this mission.

```
@SCJAllowed(javax.safetycritical.annotate.Level.SUPPORT)
@SCJPhase({
    javax.safetycritical.annotate.Phase.STARTUP,
    javax.safetycritical.annotate.Phase.INITIALIZATION})
@SCJMaySelfSuspend(true)
@SCJMayAllocate({
    javax.safetycritical.annotate.AllocationContext.CURRENT,
    javax.safetycritical.annotate.AllocationContext.INNER,
    javax.safetycritical.annotate.AllocationContext.OUTER})
protected abstract void initialize( )
```

Perform initialization of this Mission. The SCJ infrastructure calls initialize after the mission memory has been resized to match the size returned by Mission.missionMemorySize. Upon entry into initialize, the current allocation context is the mission memory area dedicated to this particular Mission.

A typical implementation of initialize instantiates and registers all ManagedSchedulable objects that constitute this Mission. The infrastructure enforces that ManagedSchedulables can only be instantiated and registered if the currently executing ManagedSchedulable is running a Mission.initialize method.

The infrastructure arranges to begin executing the registered ManagedSchedulable objects associated with a particular Mission upon return from its initialize method.

Besides initiating the associated ManagedSchedulable objects, this method may also instantiate and/or initialize mission-level data structures. Objects shared between ManagedSchedulables typically reside within the corresponding mission memory scope, but may alternatively reside in outer-nested mission memory or ImmortalMemory areas. Individual ManagedSchedulables can gain access to these objects either by supplying their references to the ManagedSchedulable constructors or by obtaining a reference to the currently running mission (from Mission.getMission), and accessing the fields or methods of this subclass.

```
@SCJAllowed(javax.safetycritical.annotate.Level.SUPPORT)
@SCJMayAllocate({})
@SCJPhase({javax.safetycritical.annotate.Phase.INITIALIZATION})
@SCJMaySelfSuspend(false)
public abstract long missionMemorySize( )
```

This method must be implemented by a safety-critical application. It is invoked by the SCJ infrastructure to determine the desired size of this Mission's mission memory area. When this method receives control, the mission memory area will include all of the backing store memory to be used for all memory areas. Therefore this method will not be able to create or call any methods that create any private memory areas. After this method returns, the SCJ infrastructure shall shrink the mission memory to a size based on the memory size returned by this method. This will make backing store memory available for the backing stores of the ManagedSchedulable objects that comprise this mission. Any attempt to introduce a new private memory area within this method will result in an OutOfMemoryError exception.

returns the required mission memory size.

```
@SCJAllowed
@SCJPhase({
   javax.safetycritical.annotate.Phase.INITIALIZATION,
   javax.safetycritical.annotate.Phase.RUN})
@SCJMayAllocate({})
@SCJMaySelfSuspend(false)
public final boolean requestTermination( )
```

This method requests termination of a mission. When this method is called, the infrastructure shall invoke signalTermination on each ManagedSchedulable object that is registered within this mission. Additionally, this method triggers the infrastructure to (1) disable all periodic event handlers associated with this

Mission so that they will experience no further releases, (2) disable all AperiodicEventHandlers so that no further releases will be honored, (3) clear the pending event (if any) for each event handler (including any OneShotEventHandlers) so that the event handler can be effectively shut down following completion of any event handling that is currently active, (4) wait for all of the ManagedSchedulable objects associated with this mission to terminate their execution, (5) invoke the ManagedSchedulable.cleanUp methods for each of the ManagedSchedulable objects associated with this mission, and (6) invoke the cleanUp method associated with this mission.

While many of these activities may be carried out asynchronously after returning from the requestTermination method, the implementation of requestTermination shall not return until all of the ManagedEventHandler objects registered with this Mission have been disassociated from this Mission so they will receive no further releases. Before returning, or at least before initialize for this same mission is called in the case that it is subsequently started, the implementation shall clear all mission state.

The first time this method is called during Mission execution, it shall return false to indicate that termination of this mission is not already in progress. Subsequent invocations of this method shall return true, and shall have no other effect.

returns false if the mission has not been requested to terminate already; otherwise returns true.

```
@SCJAllowed
@SCJPhase({
   javax.safetycritical.annotate.Phase.STARTUP,
   javax.safetycritical.annotate.Phase.INITIALIZATION,
   javax.safetycritical.annotate.Phase.RUN,
   javax.safetycritical.annotate.Phase.CLEANUP})
@SCJMayAllocate({})
@SCJMaySelfSuspend(false)
public final boolean terminationPending( )
```

Check whether the current mission is trying to terminate.

returns true if and only if this Mission's requestTermination method has been previously invoked.

G.3.14 CLASS MissionMemory

```
@SCJAllowed
public class MissionMemory extends javax.safetycritical.ManagedMemory
```

Mission memory is a linear-time scoped memory area that remains active through the lifetime of a mission. This class is final. It is instantiated by the infrastructure and entered by the infrastructure. Hence, none of its constructors are visible in the SCJ public API.

G.3.15 CLASS MissionSequencer

```
@SCJAllowed
public abstract class MissionSequencer extends
    javax.safetycritical.ManagedEventHandler
```

A MissionSequencer oversees a sequence of Mission executions. The sequence may include interleaved execution of independent missions and repeated executions of missions.

As a subclass of ManagedEventHandler, MissionSequencer's execution priority and memory budget are specified by constructor parameters.

This MissionSequencer executes vendor-supplied infrastructure code which invokes user-defined implementations of getNextMission, Mission.initialize, and Mission.cleanUp. During execution of a mission, the MissionSequencer remains blocked waiting for the mission to terminate. An invocation of signalTermination will unblock it to invoke the running mission's requestTermination method.

Constructors

```
@SCJAllowed
@SCJPhase({
   javax.safetycritical.annotate.Phase.STARTUP,
   javax.safetycritical.annotate.Phase.INITIALIZATION})
@SCJMaySelfSuspend(false)
@SCJMayAllocate({
   javax.safetycritical.annotate.AllocationContext.CURRENT,
   javax.safetycritical.annotate.AllocationContext.INNER,
   javax.safetycritical.annotate.AllocationContext.OUTER})
public MissionSequencer(PriorityParameters priority,
  ScopeParameters storage,
  ConfigurationParameters config,
  String name)
  throws java.lang.IllegalStateException
```

Construct a MissionSequencer object to oversee a sequence of mission executions.

priority — The priority at which the MissionSequencer executes.
storage — specifies the ScopeParameters for this handler
config — specifies the ConfigurationParameters for this handler

name — The name by which this MissionSequencer will be identified.
Throws IllegalStateException if invoked in an inappropriate phase.

```
@SCJAllowed
@SCJPhase({
    javax.safetycritical.annotate.Phase.STARTUP,
    javax.safetycritical.annotate.Phase.INITIALIZATION})
@SCJMaySelfSuspend(false)
@SCJMayAllocate({
    javax.safetycritical.annotate.AllocationContext.CURRENT,
    javax.safetycritical.annotate.AllocationContext.INNER,
    javax.safetycritical.annotate.AllocationContext.OUTER})
public MissionSequencer(PriorityParameters priority,
  ScopeParameters storage,
  ConfigurationParameters config)
  throws java.lang.IllegalStateException
```

This constructor behaves the same as calling MissionSequencer(priority, storage, config, null).

Methods

```
@SCJAllowed(javax.safetycritical.annotate.Level.SUPPORT)
@SCJPhase({
    javax.safetycritical.annotate.Phase.STARTUP,
    javax.safetycritical.annotate.Phase.INITIALIZATION,
    javax.safetycritical.annotate.Phase.CLEANUP})
@SCJMaySelfSuspend(false)
@SCJMayAllocate({
    javax.safetycritical.annotate.AllocationContext.CURRENT,
    javax.safetycritical.annotate.AllocationContext.INNER,
    javax.safetycritical.annotate.AllocationContext.OUTER})
protected abstract javax.safetycritical.Mission getNextMission( )
```

This method is called by infrastructure to select the initial mission to execute, and subsequently, each time one mission terminates, to determine the next mission to execute.

Prior to each invocation of getNextMission, infrastructure initializes and enters the mission memory allocation area. The getNextMission method may allocate the returned mission within this mission memory area, or it may return a reference to a Mission object that was allocated in some outer-nested mission memory area or in the ImmortalMemory area.

returns the next mission to run, or null if no further missions are to run under the control of this MissionSequencer.

```
@SCJAllowed(javax.safetycritical.annotate.Level.SUPPORT)
```

```
@Override
@SCJMaySelfSuspend(true)
@SCJMayAllocate({
   javax.safetycritical.annotate.AllocationContext.CURRENT,
   javax.safetycritical.annotate.AllocationContext.INNER,
   javax.safetycritical.annotate.AllocationContext.OUTER})
@SCJPhase({
   javax.safetycritical.annotate.Phase.STARTUP,
   javax.safetycritical.annotate.Phase.INITIALIZATION,
   javax.safetycritical.annotate.Phase.RUN,
   javax.safetycritical.annotate.Phase.CLEANUP })
public final void handleAsyncEvent( )
```

This method is used in the implementation of SCJ infrastructure. The method is not to be invoked by application code and it is not to be overridden by application code.

```
@SCJAllowed(javax.safetycritical.annotate.Level.SUPPORT)
@SCJMayAllocate({})
@SCJMaySelfSuspend(false)
@SCJPhase({
   javax.safetycritical.annotate.Phase.STARTUP,
   javax.safetycritical.annotate.Phase.INITIALIZATION,
   javax.safetycritical.annotate.Phase.RUN,
   javax.safetycritical.annotate.Phase.CLEANUP })
public final void signalTermination( )
```

Called by the infrastructure to indicate that the enclosing mission has been instructed to terminate.

The sole responsibility of this method is to call requestTermination on the currently running mission.

signalTermination will never be called by a Level 0 or Level 1 infrastructure.

G.3.16 CLASS **OneShotEventHandler**

```
@SCJAllowed(javax.safetycritical.annotate.Level.LEVEL_1)
public abstract class OneShotEventHandler extends
    javax.safetycritical.ManagedEventHandler
```

This class permits the automatic execution of time-triggered code. The handleAsyncEvent method behaves as if the handler were attached to a one-shot timer asynchronous event.

This class is abstract, non-abstract sub-classes must implement the method handleAsyncEvent and may override the default cleanUp method.

Note that the values in parameters passed to the constructors are those that will be used by the infrastructure. Changing these values after construction will have no impact on the created event handler.

Note: all time-triggered events are subject to release jitter. See section 4.8.4 for a discussion of the impact of this on application scheduling.

Constructors

```
@SCJAllowed
@SCJPhase({javax.safetycritical.annotate.Phase.INITIALIZATION})
@SCJMaySelfSuspend(false)
@SCJMayAllocate({
   javax.safetycritical.annotate.AllocationContext.CURRENT,
   javax.safetycritical.annotate.AllocationContext.INNER,
   javax.safetycritical.annotate.AllocationContext.OUTER})
public OneShotEventHandler(PriorityParameters priority,
  HighResolutionTime<?> time,
  AperiodicParameters release,
  ScopeParameters storage,
  ConfigurationParameters config,
  String name)
```

Constructs a one-shot event handler.

priority — specifies the priority parameters for this event handler. Must not be null.

time — specifies the time at which the handler should be released for execution. A relative time is relative to the start of the associated mission. An absolute time that is before the mission is started is equivalent to a relative time of 0.0. A null parameter indicates that no release of the handler should be scheduled.

release — specifies the aperiodic release parameters, in particular the deadline miss handler. A null parameter indicates that there is no deadline associated with this handler.

storage — specifies the ScopeParameters for this handler

config — specifies the ConfigurationParameters for this handler

name — a name provided by the application to be attached to this handler.

Throws IllegalArgumentException if priority or storage is null; or if time is a negative relative time; or when a deadline miss handler specfied in release is not an AperiodicEventHandler.

```
@SCJAllowed(javax.safetycritical.annotate.Level.LEVEL_1)
@SCJPhase({javax.safetycritical.annotate.Phase.INITIALIZATION})
@SCJMaySelfSuspend(false)
@SCJMayAllocate({
   javax.safetycritical.annotate.AllocationContext.CURRENT,
   javax.safetycritical.annotate.AllocationContext.INNER,
   javax.safetycritical.annotate.AllocationContext.OUTER})
```

```
public OneShotEventHandler(PriorityParameters priority,
  HighResolutionTime<?> time,
  AperiodicParameters release,
  ScopeParameters storage,
  ConfigurationParameters config)
```

This constructor behaves the same as a call to OneShotEventHandler(priority, time, release, storage, config, null).

```
@SCJAllowed(javax.safetycritical.annotate.Level.LEVEL_1)
@SCJPhase({javax.safetycritical.annotate.Phase.INITIALIZATION})
@SCJMaySelfSuspend(false)
@SCJMayAllocate({})
public OneShotEventHandler(PriorityParameters priority,
  AperiodicParameters release,
  ScopeParameters storage,
  ConfigurationParameters config)
```

This constructor behaves the same as a call to OneShotEventHandler(priority, null, release, storage, null).

Methods

```
@SCJAllowed(javax.safetycritical.annotate.Level.LEVEL_1)
@SCJPhase({
   javax.safetycritical.annotate.Phase.RUN,
   javax.safetycritical.annotate.Phase.CLEANUP})
@SCJMaySelfSuspend(false)
@SCJMayAllocate({})
public boolean deschedule( )
```

Deschedules the next release of the handler.

returns true if the handler was scheduled to be released false otherwise.

```
@SCJAllowed(javax.safetycritical.annotate.Level.LEVEL_1)
@SCJPhase({
   javax.safetycritical.annotate.Phase.RUN,
   javax.safetycritical.annotate.Phase.CLEANUP})
@SCJMaySelfSuspend(false)
@SCJMayAllocate({javax.safetycritical.annotate.AllocationContext.CURRENT})
public javax.realtime.AbsoluteTime getNextReleaseTime(AbsoluteTime dest)
```

Get the time at which this handler is next expected to be released.

dest — The instance of AbsoluteTime which will be updated in place and returned. The clock association of the dest parameter is ignored. When dest is null a new object is allocated for the result.

returns An instance of an AbsoluteTime object containing the absolute time at which this handler is expected to be released, or null if there is no currently scheduled

release. If the dest parameter is null the result is returned in a newly allocated object; otherwise it is returned in the dest object.

```
@SCJAllowed(javax.safetycritical.annotate.Level.LEVEL_1)
@SCJPhase({
   javax.safetycritical.annotate.Phase.RUN,
   javax.safetycritical.annotate.Phase.CLEANUP})
@SCJMaySelfSuspend(false)
@SCJMayAllocate({})
public void scheduleNextReleaseTime(HighResolutionTime<?> time)
```

Changes the next scheduled release time for this handler. This method can take either an AbsoluteTime or a RelativeTime for its argument, and the handler will be released as if it was created using that type for its time parameter. An absolute time in the past is equivalent to a relative time of 0.0. The rescheduling value will be effective before the return of the method.

If there is no outstanding scheduled next release, this sets one.

If scheduleNextReleaseTime is invoked with a null parameter, any next release time is descheduled.

Throws IllegalArgumentException if time is a negative RelativeTime value or clock associated with time is not the same clock that was used during construction.

G.3.17 CLASS POSIXRealtimeSignalHandler

```
@SCJAllowed(javax.safetycritical.annotate.Level.LEVEL_1)
public abstract class POSIXRealtimeSignalHandler extends
    javax.safetycritical.ManagedLongEventHandler
```

This class permits the automatic execution of code that is bound to a real-time POSIX signal. It is abstract. Concrete subclasses must implement the handleAsyncEvent method and may override the default cleanUp method. The parameter passed by the infrastructure to the handleAsyncEvent method is the id of the caught signal.

Constructors

```
@SCJAllowed(javax.safetycritical.annotate.Level.LEVEL_1)
@SCJPhase({javax.safetycritical.annotate.Phase.INITIALIZATION})
@SCJMaySelfSuspend(false)
@SCJMayAllocate({javax.safetycritical.annotate.AllocationContext.CURRENT})
public POSIXRealtimeSignalHandler(PriorityParameters priority,
  AperiodicParameters release,
  ScopeParameters storage,
  ConfigurationParameters config,
  int signalId)
```

Constructs an real-time POSIX signalt handler that will be released when the signal is delivered.

The values passed as constructor parameters are captured at construction time. Any subsequent mutation of the parameter objects has no effect on the behavior of the constructed object.

priority — specifies the priority parameters for this handle; it must not be null.
release — specifies the release parameters for this handler. A null parameter indicates that there is no deadline associated with this handler.
storage — specifies the ScopeParameters for this handler; it must not be null
config — specifies the ConfigurationParameters for this handler
signalId — specifies the id of the POSIX real-time signal that releases this handler.
Throws IllegalArgumentException when priority or storage is null; or when the signalId already has an attached handler or the signalId is outside the range of POSIX real-time signals.

Methods

```
@SCJAllowed(javax.safetycritical.annotate.Level.LEVEL_1)
@SCJMaySelfSuspend(false)
@SCJMayAllocate({})
@SCJPhase({
    javax.safetycritical.annotate.Phase.STARTUP,
    javax.safetycritical.annotate.Phase.INITIALIZATION,
    javax.safetycritical.annotate.Phase.RUN,
    javax.safetycritical.annotate.Phase.CLEANUP })
public static int getSignalId(String name)
```

Get the POSIX real-time signal id represented by name.

name — The name of the POSIX real-time signal.
returns The id of the POSIX real-time signal with this name
Throws IllegalArgumentException if there is no POSIX real-time signal with this name.

G.3.18 CLASS **POSIXSignalHandler**

```
@SCJAllowed(javax.safetycritical.annotate.Level.LEVEL_1)
public abstract class POSIXSignalHandler extends
    javax.safetycritical.ManagedEventHandler
```

This class enables the automatic execution of code that is bound to a real-time POSIX signal. It is abstract. Concrete subclasses must implement the handleAsyncEvent method and may override the default cleanUp method. The parameter passed by the infrastructure to the handleAsyncEvent method is the id of the caught signal.

Constructors

```
@SCJAllowed(javax.safetycritical.annotate.Level.LEVEL_1)
@SCJPhase({javax.safetycritical.annotate.Phase.INITIALIZATION})
@SCJMaySelfSuspend(false)
@SCJMayAllocate({javax.safetycritical.annotate.AllocationContext.CURRENT})
public POSIXSignalHandler(PriorityParameters priority,
  AperiodicParameters release,
  ScopeParameters storage,
  ConfigurationParameters config,
  int signalId)
```

Constructs a real-time POSIX signal handler that will be released when the signal is delivered.

The values passed as constructor parameters are captured at construction time. Any subsequent mutation of the parameter objects has no effect on the behavior of the constructed object.

priority — specifies the priority parameters for this handler; it must not be null.

release — specifies the release parameters for this handler. A null parameter indicates that there is no deadline associated with this handler.

storage — specifies the ScopeParameters for this handler; it must not be null

config — specifies the ConfigurationParameters for this handler

signalId — specifies the signal that releases this handler.

Throws IllegalArgumentException when priority or storage is null; or when signalIds already has an attached handler or the signalId is outside the range of POSIX signals.

Methods

```
@SCJAllowed(javax.safetycritical.annotate.Level.LEVEL_1)
@SCJMaySelfSuspend(false)
@SCJMayAllocate({})
@SCJPhase({
   javax.safetycritical.annotate.Phase.STARTUP,
   javax.safetycritical.annotate.Phase.INITIALIZATION,
   javax.safetycritical.annotate.Phase.RUN,
   javax.safetycritical.annotate.Phase.CLEANUP })
public static int getSignalId(String name)
```

Get the POSIX signal id represented by name.

name — The name of the POSIX signal.

returns The id of the POSIX signal whose name is name

Throws IllegalArgumentException if there is no POSIX signal with this name.

G.3.19 CLASS PeriodicEventHandler

```
@SCJAllowed
public abstract class PeriodicEventHandler extends
    javax.safetycritical.ManagedEventHandler
```

This class permits the automatic periodic execution of code. The handleAsyncEvent method behaves as if the handler were attached to a periodic timer asynchronous event. The handler will be executed once for every release time, even in the presence of overruns.

This class is abstract; non-abstract sub-classes must override the method handleAsyncEvent and may override the default cleanUp method.

Note that the values in parameters passed to the constructors are those that will be used by the infrastructure. Changing these values after construction will have no impact on the created event handler.

Note: all time-triggered events are subject to release jitter. See section 4.8.4 for a discussion of the impact of this on application scheduling.

Constructors

```
@SCJAllowed(javax.safetycritical.annotate.Level.LEVEL_1)
@SCJPhase({javax.safetycritical.annotate.Phase.INITIALIZATION})
@SCJMaySelfSuspend(false)
@SCJMayAllocate({
   javax.safetycritical.annotate.AllocationContext.CURRENT,
   javax.safetycritical.annotate.AllocationContext.INNER,
   javax.safetycritical.annotate.AllocationContext.OUTER})
public PeriodicEventHandler(PriorityParameters priority,
  PeriodicParameters release,
  ScopeParameters storage,
  ConfigurationParameters config,
  String name)
```

Constructs a periodic event handler.

The values passed as constructor parameters are captured at construction time. Any subsequent mutation of the parameter objects has no effect on the behavior of the constructed object.

priority — specifies the priority parameters for this periodic event handler. Must not be null.

release — specifies the periodic release parameters, in particular the start time, period and deadline miss handler. Note that a relative start time is not relative to NOW but relative to the point in time when initialization is finished and the timers are started. If the start time is absolute and it is has passed, the handler is release immediately. This argument must not be null.

storage — specifies the ScopeParameters for this periodic event handler

config — specifies the ConfigurationParameters for this periodic event handler

Throws IllegalArgumentException when priority, release, or storage is null or when any deadline miss handler specified in release is not an AperiodicEventHandler.

```
@SCJAllowed
@SCJPhase({javax.safetycritical.annotate.Phase.INITIALIZATION})
@SCJMaySelfSuspend(false)
@SCJMayAllocate({
   javax.safetycritical.annotate.AllocationContext.CURRENT,
   javax.safetycritical.annotate.AllocationContext.INNER,
   javax.safetycritical.annotate.AllocationContext.OUTER})
public PeriodicEventHandler(PriorityParameters priority,
  PeriodicParameters release,
  ScopeParameters storage,
  ConfigurationParameters config)
```

This constructor behaves the same as a call to PeriodicEventHandler(PriorityParameters, PeriodicParameters, ConfigurationParameters, String) with the arguments (priority, release, storage, config, null).

Methods

```
@SCJAllowed(javax.safetycritical.annotate.Level.LEVEL_1)
@SCJPhase({
   javax.safetycritical.annotate.Phase.RUN,
   javax.safetycritical.annotate.Phase.CLEANUP})
@SCJMayAllocate({javax.safetycritical.annotate.AllocationContext.CURRENT})
@SCJMaySelfSuspend(false)
public <T extends javax.realtime.HighResolutionTime<T>> T getActualStartTime( )
```

Get the actual start time of this handler. The actual execution start time of the handler is different from the requested start time (passed at construction time) when the requested start time is an absolute time that would occur before the mission has been started. In this case, the actual start time is the time the mission started execution. If the actual start time is equal to the effective start time, then the method behaves as if getRequestedStartTime() method has been called. If it is different, then a newly created time object is returned. The time value is associated with the same clock as that used with the original start time parameter.

returns a reference to the actual start time based on the clock used to start the timer.

```
@SCJAllowed(javax.safetycritical.annotate.Level.LEVEL_1)
@SCJPhase({
   javax.safetycritical.annotate.Phase.RUN,
   javax.safetycritical.annotate.Phase.CLEANUP})
@SCJMayAllocate({javax.safetycritical.annotate.AllocationContext.CURRENT})
```

```
@SCJMaySelfSuspend(false)
public <T extends javax.realtime.HighResolutionTime<T>> T getEffectiveStartTime( )
```

Get the effective start time of this handler. If the clock associated with the start time parameter and the period parameter (that were passed at construction time) are the same, then the method behaves as if getActualStartTime() has been called. If the two clocks are different, then the method returns a newly created object whose time is the current time of the clock associated with the period parameter (passed at construction time) when the handler is actually started.

returns a reference to a newly-created object containing the effective start time based on the clock associated with the period parameter.

```
@SCJAllowed(javax.safetycritical.annotate.Level.LEVEL_1)
@SCJPhase({
   javax.safetycritical.annotate.Phase.RUN,
   javax.safetycritical.annotate.Phase.CLEANUP})
@SCJMayAllocate({javax.safetycritical.annotate.AllocationContext.CURRENT})
@SCJMaySelfSuspend(false)
public javax.realtime.AbsoluteTime getLastReleaseTime( )
```

Get the last release time of this handler.

returns a reference to a newly-created object containing this handlers's last release time, based on the clock associated with the period parameter used at construction time.

Throws IllegalStateException if this handler has not been released since it was started.

```
@SCJAllowed(javax.safetycritical.annotate.Level.LEVEL_1)
@SCJPhase({
   javax.safetycritical.annotate.Phase.RUN,
   javax.safetycritical.annotate.Phase.CLEANUP})
@SCJMayAllocate({javax.safetycritical.annotate.AllocationContext.CURRENT})
@SCJMaySelfSuspend(false)
public javax.realtime.AbsoluteTime getNextReleaseTime( )
```

Get the time at which this handler is next expected to be released.

returns The absolute time at which this handler is expected to be released in a newly allocated AbsoluteTime object. The clock association of the returned time is the clock on which the period parameter (passed at construction time) is based.

Throws IllegalStateException if this handler has not been started or has terminated.

```
@SCJAllowed(javax.safetycritical.annotate.Level.LEVEL_1)
@SCJPhase({
   javax.safetycritical.annotate.Phase.RUN,
```

```
    javax.safetycritical.annotate.Phase.CLEANUP})
@SCJMayAllocate({javax.safetycritical.annotate.AllocationContext.CURRENT})
@SCJMaySelfSuspend(false)
public javax.realtime.AbsoluteTime getNextReleaseTime(AbsoluteTime dest)
```

Get the time at which this handler is next expected to be released.

dest — The instance of AbsoluteTime which will be updated in place and returned. The clock association of the dest parameter is ignored. When dest is null, a new object is allocated for the result.

returns The instance of AbsoluteTime passed as parameter, containing the absolute time at which this handler is expected to be released. If the dest parameter is null the result is returned in a newly allocated object. The clock association of the returned time is the clock on which the period parameter (passed at construction time) is based.

Throws IllegalStateException if this handler has not been started or has terminated.

```
@SCJAllowed(javax.safetycritical.annotate.Level.LEVEL_1)
@SCJMaySelfSuspend(false)
@SCJMayAllocate({})
@SCJPhase({
    javax.safetycritical.annotate.Phase.STARTUP,
    javax.safetycritical.annotate.Phase.INITIALIZATION,
    javax.safetycritical.annotate.Phase.RUN,
    javax.safetycritical.annotate.Phase.CLEANUP })
public <T extends javax.realtime.HighResolutionTime<T>> T getRequestedStartTime( )
```

Get the requested start time of this periodic handler. Note that the start time uses copy semantics, so changes made to the value returned by this method will not affect the start time of this handler if it has not already been started.

returns a reference to the start time parameter from the release parameters used when constructing this handler.

G.3.20 CLASS PrivateMemory

```
@SCJAllowed
public class PrivateMemory extends javax.safetycritical.ManagedMemory
```

This class cannot be directly instantiated by the application; hence there are no public constructors. Every PeriodicEventHandler is provided with one instance of PrivateMemory, its root private memory area. A schedulable object active within a private memory area can create nested private memory areas through the enterPrivateMemory method of ManagedMemory.

The rules for nested entering into a private memory are that the private memory area must be the current allocation context, and the calling schedulable object has to be the owner of the memory area. The owner of the memory area is defined to be the schedulable object that created it.

G.3.21 CLASS Services

@SCJAllowed
public class Services **extends** java.lang.Object

This class provides a collection of static helper methods.

Methods

```
@SCJAllowed
@SCJMaySelfSuspend(false)
@SCJMayAllocate({})
@SCJPhase({
   javax.safetycritical.annotate.Phase.STARTUP,
   javax.safetycritical.annotate.Phase.INITIALIZATION,
   javax.safetycritical.annotate.Phase.RUN,
   javax.safetycritical.annotate.Phase.CLEANUP })
public static javax.safetycritical.annotate.Level getComplianceLevel( )
```

Get the current compliance level of the SCJ implementation.

returns the compliance level

```
@SCJAllowed(javax.safetycritical.annotate.Level.LEVEL_1)
@SCJMaySelfSuspend(false)
@SCJMayAllocate({})
@SCJPhase({
   javax.safetycritical.annotate.Phase.STARTUP,
   javax.safetycritical.annotate.Phase.INITIALIZATION,
   javax.safetycritical.annotate.Phase.RUN,
   javax.safetycritical.annotate.Phase.CLEANUP })
public static int getDefaultCeiling( )
```

Get the default ceiling priority for objects. By default, it is PriorityScheduler.getMaxPriority. It is assumed that this can be changed using an implementation configuration option.

returns the default ceiling priority.

```
@SCJAllowed(javax.safetycritical.annotate.Level.LEVEL_1)
@SCJPhase({javax.safetycritical.annotate.Phase.INITIALIZATION})
@SCJMaySelfSuspend(false)
@SCJMayAllocate({})
public static void setCeiling(Object obj, int pri)
```

Sets the ceiling priority of object obj The priority pri can be in the software or hardware priority range. Ceiling priorities are immutable.

obj — the object who ceiling is to be set.

pri — the ceiling value.

Throws IllegalSchedulableStateException if called outside the initialization phase of a mission.

G.3.22 CLASS SingleMissionSequencer

```
@SCJAllowed
public class SingleMissionSequencer extends
      javax.safetycritical.MissionSequencer
```

Constructors

```
@SCJAllowed
@SCJMayAllocate({
    javax.safetycritical.annotate.AllocationContext.CURRENT,
    javax.safetycritical.annotate.AllocationContext.INNER,
    javax.safetycritical.annotate.AllocationContext.OUTER})
@SCJMaySelfSuspend(true)
@SCJPhase({
    javax.safetycritical.annotate.Phase.STARTUP,
    javax.safetycritical.annotate.Phase.INITIALIZATION,
    javax.safetycritical.annotate.Phase.RUN,
    javax.safetycritical.annotate.Phase.CLEANUP })
public SingleMissionSequencer(PriorityParameters priority,
  ScopeParameters storage,
  ConfigurationParameters config,
  Mission mission)
```

priority —

storage —

mission —

Methods

```
@Override
@SCJAllowed
@SCJMayAllocate({
    javax.safetycritical.annotate.AllocationContext.CURRENT,
    javax.safetycritical.annotate.AllocationContext.INNER,
    javax.safetycritical.annotate.AllocationContext.OUTER})
@SCJMaySelfSuspend(true)
@SCJPhase({
    javax.safetycritical.annotate.Phase.STARTUP,
    javax.safetycritical.annotate.Phase.INITIALIZATION,
    javax.safetycritical.annotate.Phase.RUN,
    javax.safetycritical.annotate.Phase.CLEANUP })
protected final javax.safetycritical.Mission getNextMission( )
```

see javax.safetycritical.MissionSequencer.getNextMission()

Appendix H

Javadoc Description of Package javax.safetycritical.annotate

Package Contents *Page*

H.1 Classes

H.1.1 CLASS SCJAllowed

```
@SCJAllowed
@Retention(java.lang.annotation.RetentionPolicy.CLASS)
@Target({
   java.lang.annotation.ElementType.TYPE, java.lang.annotation.ElementType.FIELD,
   java.lang.annotation.ElementType.METHOD,
   java.lang.annotation.ElementType.CONSTRUCTOR})
public @interface SCJAllowed
```

This annotation distinguishes methods, classes, and fields that may be accessed from within safety-critical Java programs. In some implementations of the safety-critical Java specification, elements which are not declared with the @SCJAllowed annotation (and are therefore not allowed in safety-critical application software) are present within the declared class hierarchy. These are necessary for full compatibility with standard edition Java, the Real-Time Specification for Java, and/or for use by the implementation of infrastructure software.

The value field equals LEVEL_0 for elements that may be used within safety-critical Java applications targeting Level 0, Level 1, or Level 2.

The value field equals LEVEL_1 for elements that may be used within safety-critical Java applications targeting Level 1 or Level 2.

The value field equals LEVEL_2 for elements that may be used within safety-critical Java applications targeting Level 2.

Absence of this annotation on a given Class, Field, Method, or Constructor declaration indicates that the corresponding element may not be accessed from within a compliant safety-critical Java application.

Attributes

```
@SCJAllowed
public boolean members () default false;
```

```
@SCJAllowed
public javax.safetycritical.annotate.Level value () default
javax.safetycritical.annotate.Level.LEVEL_0;
```

H.1.2 CLASS SCJMayAllocate

```
@SCJAllowed
```

@Retention(java.lang.annotation.RetentionPolicy.**CLASS**)
@Target({
java.lang.annotation.ElementType.TYPE, java.lang.annotation.ElementType.METHOD,
java.lang.annotation.ElementType.CONSTRUCTOR})
public @interface SCJMayAllocate

This annotation distinguishes methods that may be restricted from allocating memory in certain memory areas.

Attributes

@SCJAllowed
public javax.safetycritical.annotate.AllocationContext value () **default** {
javax.safetycritical.annotate.AllocationContext.CURRENT,
javax.safetycritical.annotate.AllocationContext.OUTER,
javax.safetycritical.annotate.AllocationContext.INNER };

H.1.3 CLASS SCJMaySelfSuspend

@SCJAllowed
@Retention(java.lang.annotation.RetentionPolicy.**CLASS**)
@Target({
java.lang.annotation.ElementType.TYPE, java.lang.annotation.ElementType.METHOD,
java.lang.annotation.ElementType.CONSTRUCTOR})
public @interface SCJMaySelfSuspend

This annotation distinguishes methods that may be restricted from blocking during execution.

Attributes

@SCJAllowed
public boolean value () **default** false;

H.1.4 CLASS SCJPhase

@SCJAllowed
@Retention(java.lang.annotation.RetentionPolicy.**CLASS**)
@Target({
java.lang.annotation.ElementType.TYPE, java.lang.annotation.ElementType.METHOD,
java.lang.annotation.ElementType.CONSTRUCTOR})
public @interface SCJPhase

This annotation distinguishes methods that may be called only from code running in a certain mission phase (e.g. Initialization or Clean Up).

Attributes

@SCJAllowed
public javax.safetycritical.annotate.Phase value () **default** {
javax.safetycritical.annotate.Phase.STARTUP,
javax.safetycritical.annotate.Phase.INITIALIZATION,
javax.safetycritical.annotate.Phase.RUN,
javax.safetycritical.annotate.Phase.CLEANUP };

The phase of the mission in which a method may run.

H.2 Interfaces

H.3 Classes

H.3.1 CLASS AllocationContext

@SCJAllowed
public enum AllocationContext

Fields

@SCJAllowed
public static final javax.safetycritical.annotate.AllocationContext CURRENT

Allocation is allowed in the current memory area.

@SCJAllowed
public static final javax.safetycritical.annotate.AllocationContext INNER

Allocation is allowed in any inner (more deeply nested) memory area.

@SCJAllowed
public static final javax.safetycritical.annotate.AllocationContext OUTER

Allocation is allowed in any outer (less deeply nested) memory area.

@SCJAllowed
public static final javax.safetycritical.annotate.AllocationContext **THIS**

Allocation is allowed in the memory area where the current object (this) was allocated.

H.3.2 CLASS Level

@SCJAllowed
public enum Level

Fields

@SCJAllowed
public static final javax.safetycritical.annotate.Level LEVEL_0

@SCJAllowed
public static final javax.safetycritical.annotate.Level LEVEL_1

@SCJAllowed
public static final javax.safetycritical.annotate.Level LEVEL_2

@SCJAllowed
public static final javax.safetycritical.annotate.Level SUPPORT

H.3.3 CLASS Phase

@SCJAllowed
public enum Phase

Fields

@SCJAllowed
public static final javax.safetycritical.annotate.Phase CLEANUP

@SCJAllowed
public static final javax.safetycritical.annotate.Phase INITIALIZATION

@SCJAllowed
public static final javax.safetycritical.annotate.Phase RUN

@SCJAllowed
public static final javax.safetycritical.annotate.Phase STARTUP

Appendix I

Javadoc Description of Package javax.safetycritical.io

Package Contents *Page*

I.1 Classes

I.2 Interfaces

I.3 Classes

I.3.1 CLASS ConnectionFactory

```
@SCJAllowed
public abstract class ConnectionFactory extends java.lang.Object
```

A factory for creating user defined connections.

Constructors

```
@SCJAllowed
@SCJMayAllocate({})
@SCJMaySelfSuspend(false)
@SCJPhase({
    javax.safetycritical.annotate.Phase.STARTUP,
    javax.safetycritical.annotate.Phase.INITIALIZATION,
    javax.safetycritical.annotate.Phase.RUN,
    javax.safetycritical.annotate.Phase.CLEANUP })
protected ConnectionFactory(String name)
```

Create a connection factory.

name — Connection name used for connection request in Connector.

Methods

```
@SCJAllowed
@SCJMaySelfSuspend(false)
@SCJMayAllocate({
    javax.safetycritical.annotate.AllocationContext.CURRENT,
    javax.safetycritical.annotate.AllocationContext.INNER,
    javax.safetycritical.annotate.AllocationContext.OUTER})
@SCJPhase({
    javax.safetycritical.annotate.Phase.STARTUP,
    javax.safetycritical.annotate.Phase.INITIALIZATION,
    javax.safetycritical.annotate.Phase.RUN,
    javax.safetycritical.annotate.Phase.CLEANUP })
public abstract javax.microedition.io.Connection create(String url)
  throws java.io.IOException, javax.microedition.io.ConnectionNotFoundException
```

Create a connection for the URL type of this factory.

url — URL for which to create the connection.

returns a connection for the URL.

Throws IOException when some other I/O problem is encountered.

```
@SCJAllowed
@Override
@SCJMayAllocate({})
@SCJMaySelfSuspend(false)
@SCJPhase({
    javax.safetycritical.annotate.Phase.STARTUP,
    javax.safetycritical.annotate.Phase.INITIALIZATION,
    javax.safetycritical.annotate.Phase.RUN,
    javax.safetycritical.annotate.Phase.CLEANUP })
public boolean equals(Object other)
```

```
@SCJAllowed
@SCJMayAllocate({})
@SCJMaySelfSuspend(false)
@SCJPhase({
    javax.safetycritical.annotate.Phase.STARTUP,
    javax.safetycritical.annotate.Phase.INITIALIZATION,
    javax.safetycritical.annotate.Phase.RUN,
    javax.safetycritical.annotate.Phase.CLEANUP })
public static javax.safetycritical.io.ConnectionFactory getRegistered(String name)
```

Get a reference to the already registered factory for a given protocol.

name — The name of the connection type.

returns The ConnectionFactory associated with the name, or null if no ConnectionFactory is registered.

```
@SCJAllowed
@SCJMayAllocate({})
@SCJMaySelfSuspend(false)
@SCJPhase({
    javax.safetycritical.annotate.Phase.STARTUP,
    javax.safetycritical.annotate.Phase.INITIALIZATION,
    javax.safetycritical.annotate.Phase.RUN,
    javax.safetycritical.annotate.Phase.CLEANUP })
public final java.lang.String getServiceName( )
```

Return the service name for a connection factory.

returns service name.

```
@SCJAllowed
@SCJMaySelfSuspend(false)
@SCJMayAllocate({
    javax.safetycritical.annotate.AllocationContext.CURRENT,
    javax.safetycritical.annotate.AllocationContext.INNER,
    javax.safetycritical.annotate.AllocationContext.OUTER})
```

```
@SCJPhase({
    javax.safetycritical.annotate.Phase.STARTUP,
    javax.safetycritical.annotate.Phase.INITIALIZATION,
    javax.safetycritical.annotate.Phase.RUN,
    javax.safetycritical.annotate.Phase.CLEANUP })
public static void register(ConnectionFactory factory)
```

Register an application-defined connection type in the connection framework. The method getServiceName specifies the protocol a factory handles. When a factory is already registered for a given protocol, the new factory replaces the old one.

factory — the connection factory.

I.3.2 CLASS ConsoleConnection

```
@SCJAllowed
public class ConsoleConnection implements
    javax.microedition.io.StreamConnection extends java.lang.Object
```

A connection for the default I/O device. The console connection can be obtained by the javax.microedition.io.Connector class with the openOutputStream method by providing ”console:” as the base url

Methods

```
@Override
@SCJAllowed
@SCJMaySelfSuspend(true)
@SCJMayAllocate({
    javax.safetycritical.annotate.AllocationContext.CURRENT,
    javax.safetycritical.annotate.AllocationContext.INNER,
    javax.safetycritical.annotate.AllocationContext.OUTER})
@SCJPhase({
    javax.safetycritical.annotate.Phase.STARTUP,
    javax.safetycritical.annotate.Phase.INITIALIZATION,
    javax.safetycritical.annotate.Phase.RUN,
    javax.safetycritical.annotate.Phase.CLEANUP })
public void close( )
```

Closes this console connection.

```
@Override
@SCJAllowed
@SCJMaySelfSuspend(false)
@SCJMayAllocate({
    javax.safetycritical.annotate.AllocationContext.CURRENT,
    javax.safetycritical.annotate.AllocationContext.INNER,
```

```
    javax.safetycritical.annotate.AllocationContext.OUTER})
@SCJPhase({
    javax.safetycritical.annotate.Phase.STARTUP,
    javax.safetycritical.annotate.Phase.INITIALIZATION,
    javax.safetycritical.annotate.Phase.RUN,
    javax.safetycritical.annotate.Phase.CLEANUP })
public java.io.InputStream openInputStream( )
```

returns the input stream for this console connection.

```
@Override
@SCJAllowed
@SCJMaySelfSuspend(false)
@SCJMayAllocate({
    javax.safetycritical.annotate.AllocationContext.CURRENT,
    javax.safetycritical.annotate.AllocationContext.INNER,
    javax.safetycritical.annotate.AllocationContext.OUTER})
@SCJPhase({
    javax.safetycritical.annotate.Phase.STARTUP,
    javax.safetycritical.annotate.Phase.INITIALIZATION,
    javax.safetycritical.annotate.Phase.RUN,
    javax.safetycritical.annotate.Phase.CLEANUP })
public java.io.OutputStream openOutputStream( )
```

returns the output stream for this console connection.

I.3.3 CLASS **SimplePrintStream**

```
@SCJAllowed
public class SimplePrintStream implements java.lang.AutoCloseable extends
    java.io.OutputStream
```

A version of OutputStream that can format a CharSequence into a UTF-8 byte sequence for writing.

Constructors

```
@SCJAllowed
@SCJMayAllocate({})
@SCJPhase({javax.safetycritical.annotate.Phase.INITIALIZATION})
@SCJMaySelfSuspend(true)
public SimplePrintStream(OutputStream stream)
```

stream — to use for output.

Methods

```
@SCJAllowed
@SCJMayAllocate({
    javax.safetycritical.annotate.AllocationContext.CURRENT,
```

```
    javax.safetycritical.annotate.AllocationContext.INNER,
    javax.safetycritical.annotate.AllocationContext.OUTER})
@SCJMaySelfSuspend(true)
@SCJPhase({
    javax.safetycritical.annotate.Phase.STARTUP,
    javax.safetycritical.annotate.Phase.INITIALIZATION,
    javax.safetycritical.annotate.Phase.RUN,
    javax.safetycritical.annotate.Phase.CLEANUP })
public boolean checkError( )
```

returns indicates whether or not an error occurred.

```
@SCJAllowed
@SCJMayAllocate({
    javax.safetycritical.annotate.AllocationContext.CURRENT,
    javax.safetycritical.annotate.AllocationContext.INNER,
    javax.safetycritical.annotate.AllocationContext.OUTER})
@SCJMaySelfSuspend(true)
@SCJPhase({
    javax.safetycritical.annotate.Phase.STARTUP,
    javax.safetycritical.annotate.Phase.INITIALIZATION,
    javax.safetycritical.annotate.Phase.RUN,
    javax.safetycritical.annotate.Phase.CLEANUP })
protected void clearError( )


@SCJAllowed
@SCJPhase({javax.safetycritical.annotate.Phase.CLEANUP})
@SCJMayAllocate({
    javax.safetycritical.annotate.AllocationContext.CURRENT,
    javax.safetycritical.annotate.AllocationContext.INNER,
    javax.safetycritical.annotate.AllocationContext.OUTER})
@SCJMaySelfSuspend(true)
public void close( )


@SCJAllowed
@SCJMayAllocate({
    javax.safetycritical.annotate.AllocationContext.CURRENT,
    javax.safetycritical.annotate.AllocationContext.INNER,
    javax.safetycritical.annotate.AllocationContext.OUTER})
@SCJMaySelfSuspend(true)
@SCJPhase({
    javax.safetycritical.annotate.Phase.STARTUP,
    javax.safetycritical.annotate.Phase.INITIALIZATION,
    javax.safetycritical.annotate.Phase.RUN,
    javax.safetycritical.annotate.Phase.CLEANUP })
public synchronized void print(CharSequence sequence)
```

The class uses the same modified UTF-8 used by java.io.DataOuputStream. There are two differences between this format and the ”standard” UTF-8 for-

mat:

1 the null byte '\\u0000' is encoded in two bytes rather than in one, so the encoded string never has any embedded nulls; and

2 only the one, two, and three byte encodings are used.

```
@SCJAllowed
@SCJMayAllocate({
   javax.safetycritical.annotate.AllocationContext.CURRENT,
   javax.safetycritical.annotate.AllocationContext.INNER,
   javax.safetycritical.annotate.AllocationContext.OUTER})
@SCJMaySelfSuspend(true)
@SCJPhase({
   javax.safetycritical.annotate.Phase.STARTUP,
   javax.safetycritical.annotate.Phase.INITIALIZATION,
   javax.safetycritical.annotate.Phase.RUN,
   javax.safetycritical.annotate.Phase.CLEANUP })
public void println( )
```

```
@SCJAllowed
@SCJMayAllocate({
   javax.safetycritical.annotate.AllocationContext.CURRENT,
   javax.safetycritical.annotate.AllocationContext.INNER,
   javax.safetycritical.annotate.AllocationContext.OUTER})
@SCJMaySelfSuspend(true)
@SCJPhase({
   javax.safetycritical.annotate.Phase.STARTUP,
   javax.safetycritical.annotate.Phase.INITIALIZATION,
   javax.safetycritical.annotate.Phase.RUN,
   javax.safetycritical.annotate.Phase.CLEANUP })
public void println(CharSequence sequence)
```

```
@SCJAllowed
@SCJMayAllocate({
   javax.safetycritical.annotate.AllocationContext.CURRENT,
   javax.safetycritical.annotate.AllocationContext.INNER,
   javax.safetycritical.annotate.AllocationContext.OUTER})
@SCJMaySelfSuspend(true)
@SCJPhase({
   javax.safetycritical.annotate.Phase.STARTUP,
   javax.safetycritical.annotate.Phase.INITIALIZATION,
   javax.safetycritical.annotate.Phase.RUN,
   javax.safetycritical.annotate.Phase.CLEANUP })
protected void setError( )
```

```
@SCJAllowed
@SCJMayAllocate({
```

```
    javax.safetycritical.annotate.AllocationContext.CURRENT,
    javax.safetycritical.annotate.AllocationContext.INNER,
    javax.safetycritical.annotate.AllocationContext.OUTER})
@SCJMaySelfSuspend(true)
@SCJPhase({
    javax.safetycritical.annotate.Phase.STARTUP,
    javax.safetycritical.annotate.Phase.INITIALIZATION,
    javax.safetycritical.annotate.Phase.RUN,
    javax.safetycritical.annotate.Phase.CLEANUP })
public void write(int b)
  throws java.io.IOException
```

Bibliography

[1] A. Burns and A. J. Wellings. *Real-Time Systems and Programming Languages:*. Addison Wesley, 4th edition, 2010.

[2] G. Bollella et. al. *The Real-Time Specification for Java*, 2000. Available from: www.rtj.org.

[3] J.J. Hunt et. al. *The Real-Time Specification for Java, V2.0*, 2014. Available from: www.rtj.org.

[4] International Electrotechnical Commission. *IEC61508. Standard for Functional Safety of Electrical/Electronic/Programmable Electronic Safety-Related Systems (E/E/PES)*, 1998.

[5] C.D. Locke. Software architecture for hard real-time applications: cyclic executives vs. fixed priority executives. *Real-Time Systems*, 4(1):37–53, 1992.

[6] RTCA. Software considerations in airborne systems and equipment certification. DO-178C, RTCA, December 2011.

[7] RTCA & European Organisation for Civil Aviation Equipment. *ED12C. Software Considerations in Airborne Systems and Equipment Certification*, January 2012.

[8] N. Storey. *Safety-Critical Computer Systems*. Addison-Wesley, 1996.

[9] United Kingdom Ministry of Defence. *Defence Standard 00-55. Requirements for Safety Related Software in Defence Equipment*, August 1997.

[10] United Kingdom Ministry of Defence. *Defence Standard 00-56. Safety Management Requirements for Defence Systems*, June 2007.