Safety-Critical Java Technology Specification

JSR-302

Version 0.93
6 December 2012
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)
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
May 2011		
November 2011		

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-178B, Level A.

This specification presents a set of Java classes providing for safety-critical application startup, 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-178B [6] or in Europe, the ED-12B [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 and careful 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 libraries. 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-178B, 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 are used throughout 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-178B and ED-12B 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			
	Catastophic	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-178B 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 carried with independence (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-178B. 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-178B) 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-178B 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-178B 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-178B Level A are also likely to able to attain certification under similar competing standards.

The use of five levels in the DO-178B 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-178B, 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 areas and defines a model based on that 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. The 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 1.1) and Java technology version 6.0. 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 a 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 by 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. Another reason for defining new classes is to control of the implementation configuration (e.g., StorageParameters) to prevent exceptions such as out of memory exceptions.

4. Annotations are defined in this specification to restrict the memory management thus enabing off-line tools to demonstrate the absence of certain run-time errors. Annotations have also been used to provide a means of documenting the assumptions made by the programmer to facilitate off-line tools in identifying 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 the

thread is 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 how it is implemented 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. The remaining chapters of this specification are normative with the exception of the Rationale section of each chapter.

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 is modeled after the ordering of chapters in the RTSJ, and 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 happenings, 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 language reference and the RTSJ. Specifically, this specification can be considered to define a subset of the Java language and the RTSJ 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 but quite limited programming model is intended to be sufficiently limited to enable certification under such standards as DO-178B Level A. This is accomplished by defining concepts such as a mission, limited startup 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 bounded set of limited schedulable objects. A schedulable object consists of a sequence of code that is scheduled by a fixed-priority Scheduler 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 until the mission is terminated, and their resources will not be reclaimed until the mission is terminated. If the application chooses to exit a mission, this specification optionally permits it to be restarted by resetting and erasing the mission memory and re-entering its initialization mode. If the application does not permit restart, 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 system, 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. 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 *cleanup phase* is entered. During the cleanup phase, each schedulable object completes its execution, and an application-defined set of cleanup methods is executed.

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) when the schedulable object completes that release. (See Section 7.1.1 for details) This scoped 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 for less complex applications.

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.

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

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.

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 implmentation 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.

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,

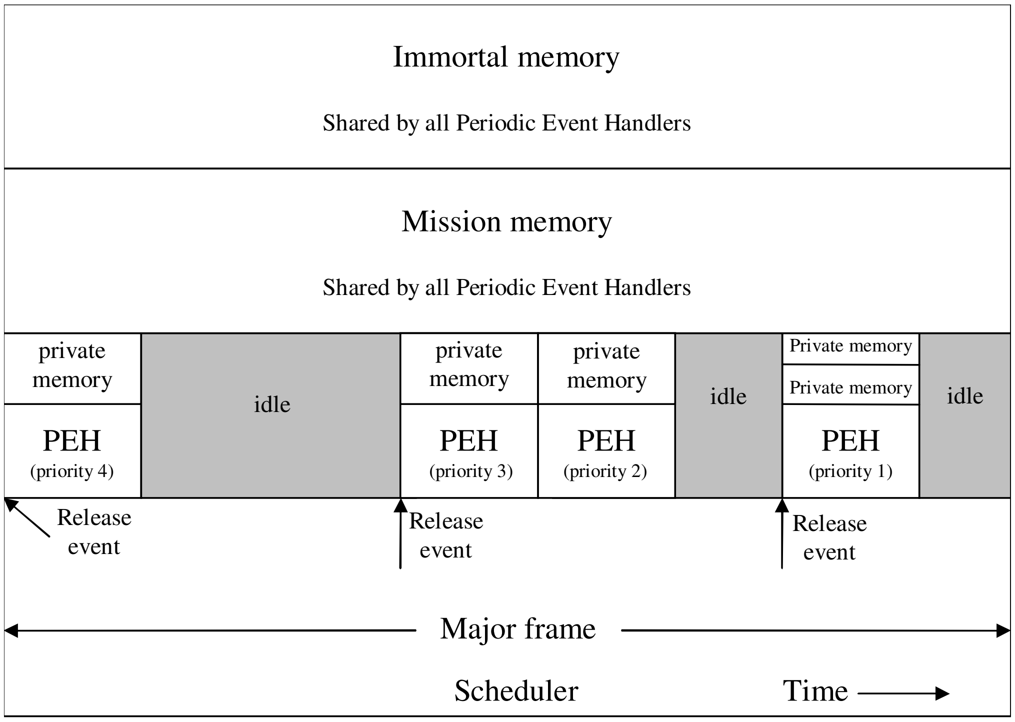

Figure 2.1: Level 0 [Cyclic Executive]

the 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 to the implementation 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 a set of PeriodicEvent-Handler (PEH) instances. 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 PrivateMemory, 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 with a set of concurrent computations, each with a priority, running under control of a fixed-priority preemptive scheduler. The computation is performed in a set 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 PrivateMemory, created for it before it is released that will be entered and exited at each release. During execution, the PEH or APEH may create, enter, and/or exit one or more other private memory areas, but these memory areas are not shared among PeriodicEventHandlers or AperiodicEventHandlers.

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

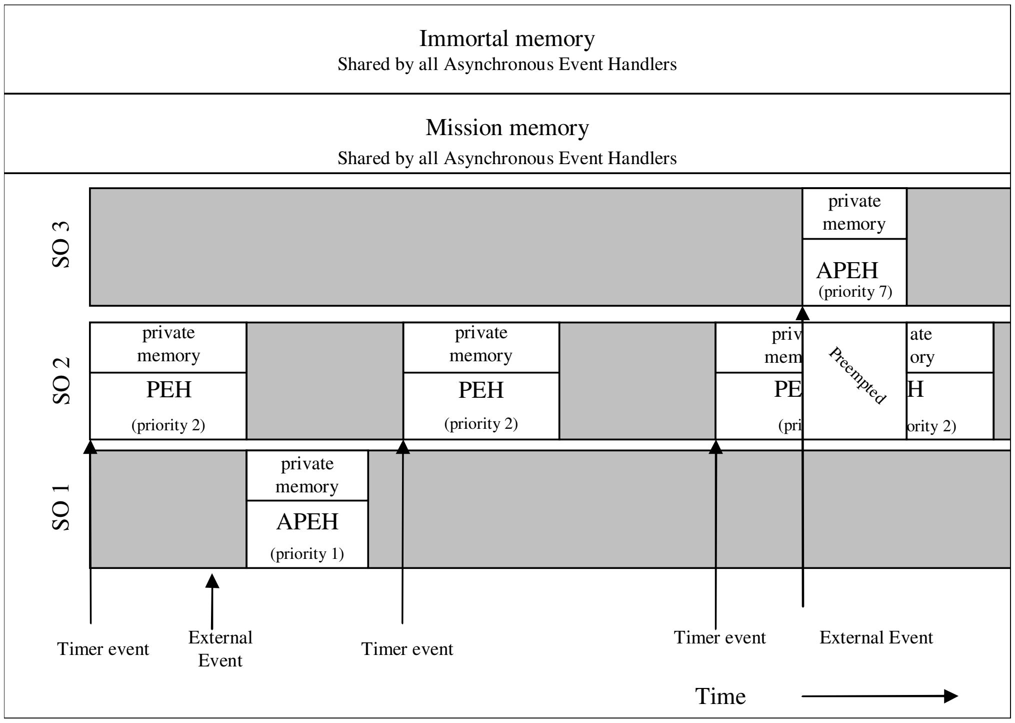

Figure 2.2: Level 1 [Single Mission]

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. Computation in a Level 2 mission is performed in a set of ManagedSchedulable (See Chapter 4.4.2 for details) objects consisting of PEHs, APEHs, and/or no-heap real-time threads. Each child mission has its own mission memory, and may also create and execute other child missions.

Each Level 2ManagedSchedulable object has a private scoped memory area created for it before its first invocation. For PEHs and APEHs, the private scoped memory area will be entered and exited at each invocation. 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 are not shared among PeriodicEventHandlers or AperiodicEventHandlers. 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 which is very similar to the NoHeapRealtimeThread defined in the RTSJ.

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 specific annotation that is pervasive in this specification is @SCJAllowed(level). Its primary use is to mark 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 reference 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 @SCJRestricted(mayAllocate = true) may contain expressions that result in memory allocation (e.g. new expressions). See Chapter 9 for details.

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:

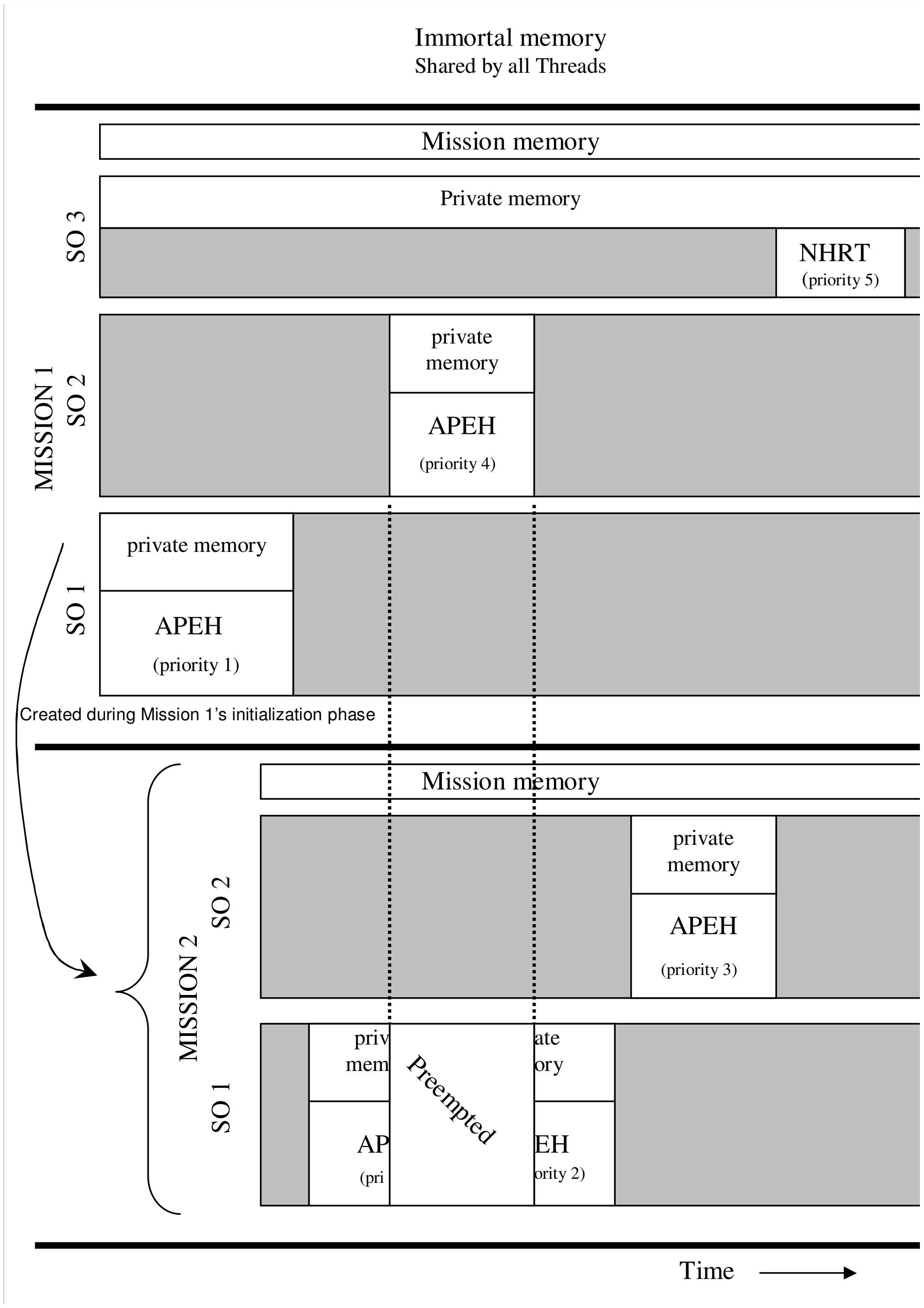

Figure 2.3: Level 2 [Nested Missions]

- Level 0: PeriodicEventHandlers.
- Level 1: PeriodicEventHandlers and AperiodicEventHandlers.
- Level 2: PeriodicEventHandlers, AsynchronousEventHandlers, 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 a 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 be available 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 supported 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-178B.

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 the ability to perform 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

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 independent threads of control, identified as ManagedSchedulables, with accompanying data structures and functional behavior.

Every SCJ application is comprised of at least one mission. Some SCJ applications are comprised of a sequence of missions, 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 running missions. Some modern safety-critical systems consist of millions of lines of code. This is far too much code to be structured as a single monolithic application. The 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 *cleanup phase*. 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 mission runs under the direction of an application-provided mission sequencer. The mission sequencer selects which mission to run next when a running mission terminates. Because the initial MissionSequencer does not belong to any mission, it resides in the ImmortalMemoryArea. This is illustrated in Figure 3.1.

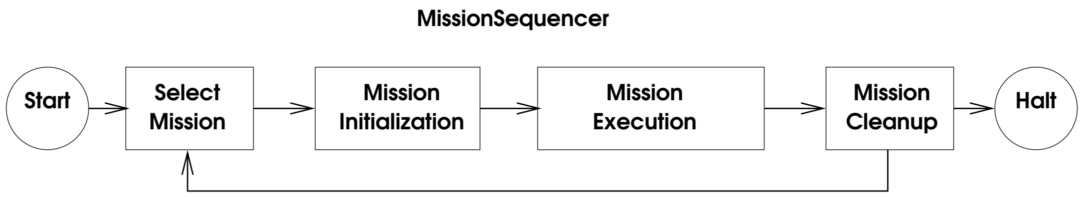

Figure 3.1: Safety-Critical Application Phases

The mission sequencer itself is structured as a ManagedSchedulable which is further defined in Chapter 4. While SCJ missions of all Levels are comprised of Managed-Schedulables, one of the key differentiating features of Level 2 missions is their ability to run inner-nested mission sequencers concurrently with the SCJ event handlers. This mechanism, which allows multiple missions to run concurrently, is supported only in Level 2 applications.

3.1.1 Application Initialization

An SCJ application is represented by a user-defined implementation of the Safelet interface. The application class that implements Safelet provides definitions of the immortalMemorySize, initializeApplication, and getSequencer methods. The infrastructure immortalMemorySize, initializeApplication, and getSequencer in this sequence with ImmortalMemoryArea as the current allocation context. The application may allocate global data structures within the initializeApplication method. The getSequencer method returns a reference to the MissionSequencer that oversees execution of the application. The MissionSequencer identifies a sequence of one or more user-defined missions, each of which is represented by a class extending the Mission class.

3.1.2 Mission Initialization

The MissionSequencer invokes the initialize method associated with each mission. The initialize instantiates and registers all of the ManagedSchedulable objects contained in each mission and allocates and initializes all objects that will be shared among these ManagedSchedulable objects. All ManagedSchedulable objects used by the mission shall be registered by initialize.

All ManagedSchedulable objects shall be allocated in the MissionMemory area of the mission to which they belong. This shall be enforced at registration time by the register method throwing an IllegalArgumentException. Also, each mission object shall reside in either the same scope as its MissionSequencer or the MissionMemory held by the MissionSequencer.

3.1.3 Mission Execution

Upon return from the initialize method, the SCJ infrastructure starts up each of the threads that are bound to the ManagedSchedulable objects that were registered by the initialize method.

For each managed schedulable (with the exception of some interrupt handlers and the MissionSequencer), the SCJ infrastructure provides a PrivateMemory area which serves as the default memory area to hold the thread's temporary memory allocations. Each managed schedulable is free to introduce additional inner-nested PrivateMemory areas to hold temporary objects that have shorter lifetimes than the duration of each release. Individual managed schedulables may also arrange to allocate objects in ImmortalMemoryArea or in outer-nested MissionMemory areas.

Mission execution continues until all of the threads bound to the mission's managed schedulables terminate. A ManagedThread terminates by simply returning from its run() method. The only way to terminate ManagedEventHandlers is to invoke the corresponding mission's requestTermination method. This arranges that each ManagedEventHandler thread will terminate following completion of its current event handling activities.

3.1.4 Mission Cleanup

The application defines the cleanup method. The SCJ infrastructure invokes cleanup after all of the managed schedulables associated with this mission have terminated their execution and their corresponding cleanup methods have been invoked for each of the terminated managed schedulables.

The cleanup phase 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.

3.2 Semantics and Requirements

An application consists of one or more missions executed sequentially or concurrently, as initiated by a user-defined implementation of the Safelet interface. Each

Mission has its own MissionMemory which holds objects representing the Mission state. The independent threads and event handling activities that comprise the Mission communicate with each other by modifying the shared objects that reside within the MissionMemory.

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

3.2.1 Class Initialization

After loading all application classes into immortal memory, class initialization shall be performed by the implementation. An SCJ application shall have no cyclic dependencies among the class initialization methods for the classes that comprise it. The SCJ infrastructure shall initialize all of the classes that comprise the application in an order determined by topological sort of class interdependencies before performing Safelet initialization. For purposes of the dependency analysis, conforming implementations shall 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 a new 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, such an 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 the dependency analysis ignores control-flow considerations,

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

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 on the class that is being initialized.

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 described below.

- The Safelet object is allocated within the ImmortalMemoryArea area.
- The Safelet's immortalMemorySize method is invoked to determine the desired size of ImmortalMemoryArea. If the actual size of the remaining ImmortalMemoryArea is smaller than the value returned from immortalMemorySize, Safelet initialization immediately aborts.
- Infrastructure invokes the Safelet's initializeApplication method to allow the application to allocate global data structures in the ImmortalMemoryArea.
- Infrastructure invokes the Safelet's getSequencer method, with the ImmortalMemoryArea area as the current allocation context. The value returned represents the MissionSequencer that runs this application. If null is returned, the application immediately halts.
- The application-specific implementation of Safelet.getSequencer 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 the application-specific implementation of Safelet.getSequencer that are not handled within that implementation shall follow the same rules for propagation and handling described in Chapter 11 for application methods. Storage space for such exceptions shall be included in the application's mission memory StorageParameters parameters.
- The mechanism to specify the StorageParameters for the Safelet initialization thread is implementation defined.
- The mechanism to specify the priority at which the Safelet initialization thread

runs relative to other non-Java threads which may be running concurrently on the same hardware is implementation defined.

- It is implementation-defined how much total memory is available within the SCJ run-time environment to represent the combined total of immortal memory, StorageParameters and stack memory requests for the Safelet initialization thread, and StorageParameters and stack memory requests for all other ManagedSchedulables that comprise the application. Note that whether the StorageParameters request includes a representation of the corresponding thread's run-time stack memory requirements is also implementation-defined. Whether the memory used to satisfy each ManagedSchedulable's StorageParameters and stack memory requests 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 the stated order:

- Infrastructure code creates and starts up the MissionSequencer by starting its bound thread. The memory resources specified by the StorageParameters argument to the MissionSequencer's constructor are set aside at the time that the infrastructure starts the MissionSequencer's bound thread.
- Next, infrastructure code fires the MissionSequencer's associated event.
- 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]
- In the case that a MissionSequencer nests within a Level 2Mission, the MissionSequencer must be registered during execution of that Mission's initialize code. The thread's storage resource requirements are specified by the StorageParameters argument to the MissionSequencer constructor. These resources are reserved for execution of the MissionSequencer at the time the MissionSequencer's bound thread is started, following return from the enclosing Mission's initialize method.
- When the MissionSequencer begins to execute, it instantiates a MissionMemory object to hold data corresponding to the missions that are to be executed by this MissionSequencer. The backing store associated with this MissionMemory object is initially sized to represent all of the remaining backing store memory associated with this MissionSequencer's bound thread.

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

- Next, the MissionSequencer enters the newly created MissionMemory 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 MissionMemory area, or it may return a Mission object that resides in some memory area that is more outer-nested than the MissionMemory area.
- Upon return from getNextMission, the MissionSequencer invokes the missionMemorySize method on the returned mission object and truncates the current MissionMemory area to the size returned from the missionMemorySize method. If the value returned from missionMemorySize is larger than the size of the current MissionMemory, the MissionSequencer aborts the current mission, exits the current MissionMemory, reclaiming the memory of all objects allocated within it, and endeavors to replace the current mission with a new mission by reinvoking its own getNextMission method.
- When the size of MissionMemory is truncated, the surplus memory that had previously been part of the current MissionMemory area's backing store is returned to the pool of backing store memory available for new ManagedMemory areas to be associated with the MissionSequencer thread, or with sub-threads spawned by one of its inner-nested missions.
- After successfully resizing MissionMemory, the MissionSequencer thread invokes the selected Mission object's initialize method.
- Upon return from initialize, the MissionSequencer thread starts all of the managed schedulables that were registered by the initialize method. Most of these threads are started in a blocked state, awaiting periodic or asynchronous firings of the corresponding events. The MissionSequencer thread then blocks itself, awaiting mission termination.
- In general, the management of memory to satisfy the StorageParameters request specified by the arguments of a MissionSequencer's constructor is implementation defined. The implementation-defined memory management technique for the outer-most MissionSequencer's StorageParameters request may be different than the implementation-defined 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:

- With MissionMemory 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 objects that will run as part of this mission. A StorageParameters object shall be associated with each ManagedSchedulable at construction time. This StorageParameters object shall describe the resources required for execution of the corresponding bound thread. Reservation of the requested resources is made at the time the bound thread is started rather than at the time the ManagedSchedulable object is constructed. It is common, but not necessary, for all of this mission's ManagedSchedulable objects to be allocated within MissionMemory by the initialize method. If a ManagedSchedulable to be associated with this mission is not allocated in this mission's MissionMemory, then it must reside in some more outer-nested memory area.

- During its execution, the initialize method may also allocate mission-relevant data structures in MissionMemory. These data structures may be shared between the managed schedulables that comprise this mission. The initialize method may also use a stack of nested PrivateMemory areas to hold temporary objects relevant to its computations.
- Upon return from initialize, the infrastructure invokes the mission's getSchedule method in a Level 0 run-time environment. The infrastructure creates an array representing all of the ManagedSchedulable objects that were registered by the initialize method and passes this array as an argument to the mission's getSchedule method. To provide predictable application behavior, the entries within this array are sorted in the same order that the initialize method registered the respective ManagedSchedulable objects. This array resides in MissionMemory, but is used exclusively for the purpose of communicating with the getSchedule method. If 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.
- As each of the ManagedSchedulable's bound threads is started, the infrastructure reserves for the bound thread the memory resources requested by the corresponding ManagedSchedulable object's StorageParameters object. The backing store for each independent thread is obtained by setting aside portions of the backing store memory that had been previously associated with the MissionSequencer's event handling thread. 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 PrivateMemory areas shall not fail due to fragmentation.
- The MissionSequencer object's event handling thread then waits for the Mission's execution to terminate. This event handling thread shall remain in a blocked state until the requestTermination method is called. Once termination has been requested, the MissionSequencer object's event handling thread shall complete the Mission termination sequence. This termination sequence shall

require the use of implementation-defined resources that must be accounted for by applications designed to provide highly predictable timing behavior.

- When the MissionSequencer's event handling thread detects that the Mission's mission phase has terminated, by confirming that all of the threads bound to the mission's ManagedSchedulable objects have terminated, it arranges to execute the cleanup method corresponding to each of those ManagedSchedulables.
- All ManagedSchedulablecleanup methods shall be called by the MissionSequencer's event handling thread. The calls to the cleanup methods shall not be performed until all of the ManagedSchedulables have terminated. When the cleanup method is called, a private memory area shall be provided for its use, and shall be the current memory area. If desired, the cleanup method may introduce a new PrivateMemory area. The Memory allocated to ManagedSchedulables shall be available to be freed when each Mission's cleanup method returns. If an exception is thrown in a cleanup method and not caught in the method, it shall be caught and ignored by the MissionSequencer's event handling thread.
- After the cleanUp methods for all of the mission's ManagedSchedulable objects have been executed, the MissionSequencer thread invokes the Mission's cleanup method.
- After the mission finishes its execution, including the mission cleanup code, control exits the MissionMemory area allocated above, releasing all of the objects that had been allocated within that MissionMemory, including the Mission object itself if it was allocated within the MissionMemory 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 getSchedule method. This method returns a reference to a CyclicSchedule object which represents the static cyclic schedule for the PEH objects 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 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<MissionLevel **extends** Mission>

Description

A safety-critical application consists of one or more missions, executed concurrently or in sequence. Every safety-critical application is represented by an implementation of Safelet which identifies the outer-most MissionSequencer. This outer-most MissionSequencer takes responsibility for running 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.

For the MissionSequencer returned from getSequencer, the SCJ infrastructure arranges for an independent thread to begin executing the code for that sequencer and then waits for that thread to terminate its execution.

Methods

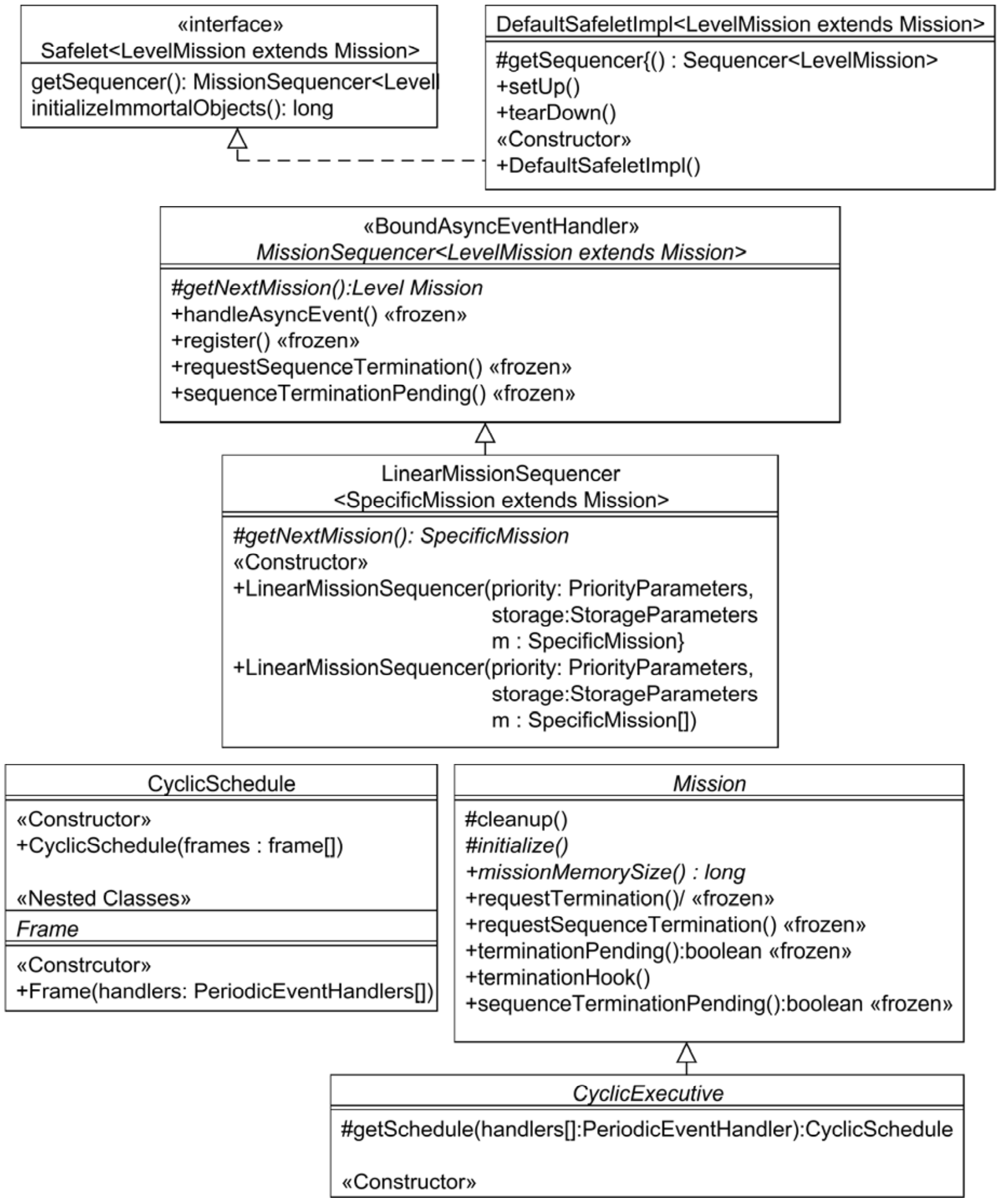
«interface»
Safelet<LevelMission extends Mission>
getSequencer(): MissionSequencer<Level
initializeImmortalObjects(): long
DefaultSafeletImpl<LevelMission extends Mission>
#getSequencer{() : Sequencer<LevelMission>
+setUp()
+tearDown()
«Constructor»
+DefaultSafeletImpl()
«BoundAsyncEventHandler»
MissionSequencer<LevelMission extends Mission>
#getNextMission():Level Mission
+handleAsyncEvent() «frozen»
+register() «frozen»
+requestSequenceTermination() «frozen»
+sequenceTerminationPending() «frozen»
LinearMissionSequencer
<SpecificMission extends Mission>
#getNextMission(): SpecificMission
«Constructor»
+LinearMissionSequencer(priority: PriorityParameters, storage:StorageParameters m : SpecificMission}
+LinearMissionSequencer(priority: PriorityParameters, storage:StorageParameters m : SpecificMission[])
CyclicSchedule
«Constructor»
+CyclicSchedule(frames : frame[])
«Nested Classes»
Frame
«Constrcutor»
+Frame(handlers: PeriodicEventHandlers[])
Mission
#cleanup()
#initialize()
+missionMemorySize() : long
+requestTermination()/ «frozen»
+requestSequenceTermination() «frozen»
+terminationPending():boolean «frozen»
+terminationHook()
+sequenceTerminationPending():boolean «frozen»
CyclicExecutive
#getSchedule(handlers[]:PeriodicEventHandler):CyclicSchedule
«Constructor»

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

@SCJAllowed(javax.safetycritical.annotate.Level.SUPPORT)
@SCJRestricted({javax.safetycritical.annotate.Phase.INITIALIZATION})
public
javax.safetycritical.MissionSequencer<MissionLevel> 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)
public long immortalMemorySize( )

returns the amount of additional immortal memory that must be available for the immortal memory allocations to be performed by this application. If the amount of memory remaining in immortal memory is less than this requested size, the infrastructure halts execution of the application upon return from this method.

@SCJAllowed(javax.safetycritical.annotate.Level.SUPPORT)
@SCJRestricted({javax.safetycritical.annotate.Phase.INITIALIZATION})
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 that are in immortal memory. initializeApplication shall be invoked after immortalMemorySize, and before getSequencer.

3.4.2 javax.safetycritical.MissionSequencer

Declaration

@SCJAllowed
public abstract class MissionSequencer<SpecificMission **extends** Mission>

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 is bound to an event handling thread. The bound thread's execution priority and memory budget are specified by constructor parameters.

This MissionSequencer executes vendor-supplied infrastructure code which invokes user-defined implementations of MissionSequencer. getNextMission, Mission.initialize, and Mission.cleanUp. During execution of an inner-nested mission, the MissionSequencer's thread remains blocked waiting for the mission to terminate. An invocation of MissionSequencer.request SequenceTermination will unblock this waiting thread so that it can perform an invocation of the running mission's requestTermination method if the mission is still running and its termination has not already been requested.

Note that if a MissionSequencer object is preallocated by the application, it must be allocated in the same scope as its corresponding Mission.

Constructors

```
@SCJAllowed
@SCJRestricted({javax.safetycritical.annotate.Phase.INITIALIZATION})
public MissionSequencer(PriorityParameters priority,
  StorageParameters storage,
  String name)
  throws java.lang.IllegalStateException
```

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

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.

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

Throws IllegalStateException if invoked at an inappropriate time. The only appropriate times for instantiation of a new MissionSequencer are (a) during execution of Safelet.getSequencer by SCJ infrastructure during startup of an SCJ application, and (b) during execution of Mission.initialize by SCJ infrastructure during initialization of a new mission in a Level 2 configuration of the SCJ run-time environment. Note that the static checker for SCJ forbids instantiation of MissionSequencer objects outside of mission initialization, but it does not prevent Mission.initialize in a Level 1 application from attempting to instantiate a MissionSequencer.

Memory behavior: This constructor requires that the "priority" argument reside in a scope that encloses the scope of the "this" argument. This constructor requires that the "storage" argument reside in a scope that encloses the scope of the "this" argument.

This constructor requires that the "name" argument reside in a scope that encloses the scope of the "this" argument.

```
@SCJAllowed
@SCJRestricted({javax.safetycritical.annotate.Phase.INITIALIZATION})
public MissionSequencer(PriorityParameters priority,
  StorageParameters storage)
  throws java.lang.IllegalStateException
```

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

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.

Throws IllegalStateException if invoked at an inappropriate time. The only appropriate times for instantiation of a new MissionSequencer are (a) during execution of Safelet.getSequencer by SCJ infrastructure during startup of an SCJ application, and (b) during execution of Mission.initialize by SCJ infrastructure during initialization of a new mission in a Level 2 configuration of the SCJ run-time environment. Note that the static checker for SCJ forbids instantiation of MissionSequencer objects outside of mission initialization, but it does not prevent Mission.initialize in a Level 1 application from attempting to instantiate a MissionSequencer.

Memory behavior: This constructor requires that the "priority" argument reside in a scope that encloses the scope of the "this" argument. This constructor requires that the "storage" argument reside in a scope that encloses the scope of the "this" argument.

Methods

```
@SCJAllowed(javax.safetycritical.annotate.Level.SUPPORT)
protected abstract SpecificMission 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 instantiates and enters the MissionMemory allocation area. The getNextMission method may allocate the returned mission within this newly instantiated MissionMemory allocation area, or it may return a reference to a Mission object that was allocated in some outer-nested MissionMemory 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
public final void requestSequenceTermination( )

Initiate mission termination by invoking the currently running mission's requestTermination method. Upon completion of the currently running mission, this MissionSequencer shall return from its handleAsyncEvent method without invoking getNextMission and without starting any additional missions. Its handleAsyncEvent method will not be invoked again.

Note that requestSequenceTermination does not force the sequence to terminate because the currently running mission must voluntarily relinquish its resources.

Control shall not return from requestSequenceTermination until after the requestTermination method for this mission sequencer's currently running mission has been invoked and control has returned from that invocation. The running mission's requestTermination method is invoked by the requestSequenceTermination method.

It is implementation-defined whether Mission.requestTermination has been called when requestSequenceTermination returns.

@SCJAllowed
public final boolean sequenceTerminationPending( )

returns true if and only if the requestSequenceTermination method has been invoked for this MissionSequencer object.

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
public Mission( )

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 he 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.

Memory behavior: This constructor may allocate objects within the same MemoryArea that holds the implicit this argument.

Methods

@SCJAllowed(javax.safetycritical.annotate.Level.SUPPORT)
protected void cleanUp( )

Method to clean data structures and machine state upon termination of this Mission's execute phase. Infrastructure code running in the controlling MissionSequencer's bound thread invokes cleanUp after all ManagedSchedulables associated with this Mission have terminated, but before control leaves the corresponding MissionMemory area. The default implementation of cleanUp does nothing.

@SCJAllowed
public static javax.safetycritical.Mission getCurrentMission( )

Obtain the current mission.

returns the instance of the Mission to which the currently executing Managed-Schedulable corresponds. The current Mission is known from the moment when initialize has been invoked and continues to be known until the mission's last cleanUp method has been completed. Otherwise, returns null.

@SCJAllowed
public javax.safetycritical.MissionSequencer> getSequencer( )

returns the MissionSequencer that is overseeing execution of this mission.

@SCJAllowed(javax.safetycritical.annotate.Level.SUPPORT)
protected abstract void initialize( )

Perform initialization of this Mission. Infrastructure calls initialize after the Mission has been instantiated and the MissionMemory has been resized to match the size returned from Mission.missionMemorySize. Upon entry into the initialize method, the current allocation context is the MissionMemory area dedicated to this particular Mission.

The default implementation of initialize does nothing.

A typical implementation of initialize instantiates and registers all Managed-Schedulable 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 under the direction of the SCJ infrastructure. 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 certain mission-level data structures. Note that objects shared between ManagedSchedulables typically reside within the corresponding MissionMemory scope, but may alternatively reside in outer-nested MissionMemory 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 (the value returned from Mission.getCurrentMission), coercing the reference to the known Mission subclass, and accessing the fields or methods of this subclass that represent the shared data objects.

@SCJAllowed(javax.safetycritical.annotate.Level.SUPPORT)
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

MissionMemory area. When this method receives control, the MissionMemory 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 PrivateMemory areas. After this method returns, the SCJ infrastructure shall shrink the MissionMemory 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 PrivateMemory area within this method will result in an OutOfMemoryError exception.

@SCJAllowed
public final void requestTermination( )

This method provides a standard interface for requesting termination of a mission. Once this method is called during Mission execution, subsequent invocations of terminationPending shall return true, shall invoke this object's terminationHook method, and shall invoke requestSequenceTermination on each inner-nested MissionSequencer object that is registered for execution within this mission. Additionally, this method has the effect of arranging to (1) disable all periodic event handlers associated with this Mission so that they will experience no further firings, (2) disable all AperiodicEventHandlers so that no further firings will be honored, (3) clear the pending event (if any) for each event handler 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 invoking 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 after 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.

@SCJAllowed(javax.safetycritical.annotate.Level.SUPPORT)
protected void terminationHook( )

This method shall be invoked by requestTermination. Application-specific subclasses of Mission may override the terminationHook method to supply application-specific mission shutdown code.

3.4.4 javax.safetycritical.Frame

Declaration

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

Constructors

@SCJAllowed
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. Under normal circumstances, this Frame object is instantiated within the MissionMemory area that corresponds to the Level0Mission that is to be scheduled.

Within each execution frame of the CyclicSchedule, the PeriodicEventHandler objects represented by the handlers array will be fired in same order as they appear within this array. Normally, PeriodicEventHandlers are sorted into decreasing priority order prior to invoking this constructor.

Memory behavior: This constructor may allocate objects within the same MemoryArea that holds the implicit this argument.

This constructor requires that the "duration" argument reside in a scope that encloses the scope of the "this" argument. This constructor requires that the "handlers" argument reside in a scope that encloses the scope of the "this" argument.

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
public 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.

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

Memory behavior: This constructor may allocate objects within the same MemoryArea that holds the implicit this argument.

This constructor requires that the "frames" argument reside in a scope that encloses the scope of the "this" argument.

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
public CyclicExecutive( )

Construct a CyclicExecutive object.

Methods

@SCJAllowed(javax.safetycritical.annotate.Level.SUPPORT)
public 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 override the getSchedule 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 MissionMemory area shall be the active allocation context. The value returned from getSchedule must reside in the current mission's MissionMemory 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.

Memory behavior: This constructor requires that the "handlers" argument reside in a scope that encloses the scope of the "this" argument.

3.4.7 LinearMissionSequencer

Declaration

@SCJAllowed
public class LinearMissionSequencer<SpecificMission>

extends javax.safetycritical.MissionSequencer<SpecificMission>

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 <SpecificMission> 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
@SCJRestricted({javax.safetycritical.annotate.Phase.INITIALIZATION},
   maySelfSuspend = false)
public LinearMissionSequencer(PriorityParameters priority,
  StorageParameters storage,
  boolean repeat,
  SpecificMission m)
  throws java.lang.IllegalArgumentException,
      java.lang.IllegalStateException
```

Construct a LinearMissionSequencer object to oversee execution of the 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.

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

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

Throws IllegalArgumentException if any of the arguments equals null.

Memory behavior: This constructor requires that the "priority" argument reside in a scope that encloses the scope of the "this" argument. This constructor requires that the "storage" argument reside in a scope that encloses the scope of the "this" argument. This constructor requires that the "m" argument reside in a scope that encloses the scope of the "this" argument.

```
@SCJAllowed
@SCJRestricted({javax.safetycritical.annotate.Phase.INITIALIZATION},
   maySelfSuspend = false)
public LinearMissionSequencer(PriorityParameters priority,
  StorageParameters storage,
  boolean repeat,
  SpecificMission m,
  String name)
  throws java.lang.IllegalArgumentException,
      java.lang.IllegalStateException
```

Construct a LinearMissionSequencer object to oversee execution of the 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.

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

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

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.

Memory behavior: This constructor requires that the "priority" argument reside in a scope that encloses the scope of the "this" argument. This constructor requires that the "storage" argument reside in a scope that encloses the scope of the "this" argument. This constructor requires that the "m" argument reside in a scope that encloses the scope of the "this" argument. This constructor requires that the "name" argument reside in a scope that encloses the scope of the "this" argument.

```
@SCJAllowed
@SCJRestricted({javax.safetycritical.annotate.Phase.INITIALIZATION},
   maySelfSuspend = false)
public LinearMissionSequencer(PriorityParameters priority,
  StorageParameters storage,
  SpecificMission [] missions,
  boolean repeat)
  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.

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. It is required 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.

Throws IllegalArgumentException if any of the arguments equals null.

Memory behavior: This constructor requires that the "priority" argument reside in a scope that encloses the scope of the "this" argument. This constructor requires that the "storage" argument reside in a scope that encloses the scope of the "this" argument.

```
@SCJAllowed
@SCJRestricted({javax.safetycritical.annotate.Phase.INITIALIZATION},
    maySelfSuspend = false)
public LinearMissionSequencer(PriorityParameters priority,
  StorageParameters storage,
  SpecificMission [] 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.

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.

Memory behavior: This constructor requires that the "priority" argument reside in a scope that encloses the scope of the "this" argument. This constructor requires that the "storage" argument reside in a scope that encloses the scope of the "this" argument. This constructor requires that the "name" argument reside in a scope that encloses the scope of the "this" argument.

Methods

```
@SCJAllowed(javax.safetycritical.annotate.Level.SUPPORT)
@SCJRestricted({javax.safetycritical.annotate.Phase.INITIALIZATION},
   maySelfSuspend = false)
@Override
protected final SpecificMission 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 Also: 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 startup 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.

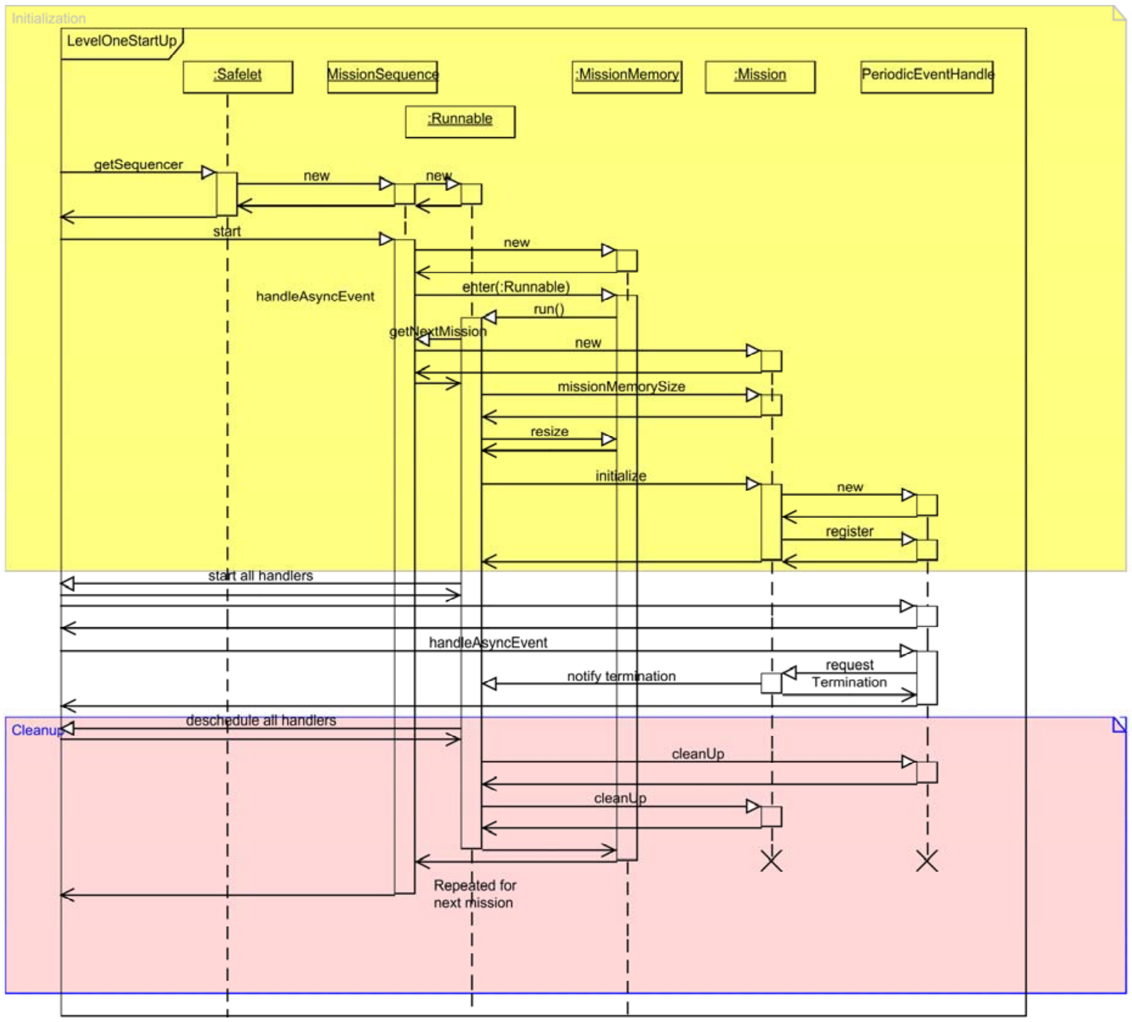

Figure 3.3: Sample Level 1 startup sequence diagram

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 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-178B 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. 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's thread, it is important that the MissionSequencer constructors allow specification of the corresponding bound thread's StorageParameters. In the case that the StorageParameters 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 MissionMemory. For simpler applications, it may be desirable for the Mission object to reside in memory that nests external to the corresponding MissionMemory area. Note, for example, that the APIs for the LinearMissionSequencer and RepeatingMissionSequencer data types require that all of the missions to be executed by these mission sequencers be passed in as constructor arguments. Thus, these missions must reside in a memory area that is external to the MissionMemory area that will correspond to the mission itself.

In more complex systems, it may be preferable to allocate the mission object within its own MissionMemory 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 by a MissionSequencer following termination of a prior execution. 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 MissionMemory 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 MissionMemory 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 MissionMemory area, the MissionMemory area is allocated and initialized before the Mission object is allocated. Thus, at the time MissionMemory 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 MissionMemory area to include all available backing store memory associated with the MissionSequencer's currently executing thread. Upon return from the getNextMission method, the infrastructure invokes the returned Mission object's missionMemorySize method and then truncates the MissionMemory 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 PrivateMemory 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. In earlier drafts of the SCJ specification, the specification described a hierarchical decomposition of all memory that described for each mission how all of the memory dedicated to that mission could be divided into smaller portions, each dedicated to the reliable execution of one of the mission's managed schedulables. It was then thought that associating three contiguous regions of memory for the dedicated use of each SCJ thread 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 thread, and described the decomposition of the memory associated with a mission sequencer's thread into independent memory segments to represent the needs of each of the currently running mission's managed schedulable threads.

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 StorageParameters object associated with every managed schedulable addresses only the hiearchical decomposition of backing store memory for threads. 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 threads that comprise a particular mission, the programmer must decide between several alternative mechanisms for providing the individual threads with references to the shared data structures.

- The individual threads may invoke javax.safetycritical.Mission.getCurrent Mission, coerce the result to the known Mission subclass, and directly access the fields and methods of this known 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 mission's relevant managed schedulable objects.

Though the software engineering tradeoffs must be assessed by developers in each specific context, it is generally recommended that the latter approach is the 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 that information is shared only with other components that truly need access to that information. Third, the implementation of individual threads can be made more independent of the mission within which they run. Since the thread doesn't need to know the type of the mission that hosts its execution, the implementation of the thread 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 schedulables, termination of the mission is fairly automatic. When application code invokes the mission's requestTermination method, the infrastructure arranges to disable all further firings of the corresponding events. All of the managed schedulables associated with the mission will terminate upon completion of any currently running event handlers.

Termination of Level 2 missions that include the execution of ManagedThread schedulables may be 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 an opportunity to provide application-specific code that will automatically be invoked by the infrastructure each time a mission's requestTermination method is invoked. In the implementation of a Mission subclass, programmers are free to override the implementation of the terminationHook method.

Typically, an application-specific implementation of terminationHook will include code that sets certain mission state variables which the mission's running threads occassionally query. When the thread discovers by examining these state variables that mission termination has been requested, the thread performs whatever cleanup activities it considers to be essential and returns, thus terminating its execution.

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

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 PriorityParameters 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 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 run-time 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 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 generally considered good practice for developers of Level 0 missions to use Java synchronized methods to access all data shared between multiple periodic event handlers even though such synchronization is not strictly required when the mission executes within a Level 0 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 sequencer's requestSequenceTermination 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 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 some thread invokes the mission sequencer's requestSequenceTermination method or the mission's requestTermination method.

3.6.9 Example of a Static Level 0 Application

This section provides an example implementation 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.safetycritical.CyclicExecutive;
import javax.safetycritical.CyclicSchedule;
import javax.safetycritical.LinearMissionSequencer;
import javax.safetycritical.MissionSequencer;
import javax.safetycritical.PeriodicEventHandler;
import javax.safetycritical.StorageParameters;
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 StorageParameters(10000, null),
      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.RelativeTime;
import javax.safetycritical.PeriodicEventHandler;
import javax.safetycritical.StorageParameters;
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 StorageParameters(10000, null), 0);
     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 MissionMemory.
   *
   * 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 MissionMemory. A more frugal
      // implementation would allocate these objects in PrivateMemory.
      //
      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 ImmortalMemoryArea. The example described in this section allocates the same SimpleCyclicExecutive object in MissionMemory. For illustrative purposes, this example repeatedly executes the SimpleCyclicExecutive mission. Each time the SimpleCyclicExecutive mission terminates, the MissionMemory 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 implementations of LinearMissionSequencer and RepeatingMissionSequencer found in the SCJ library require that the sequenced missions reside external to the MissionMemory area. In order to arrange for Mission objects to be newly allocated within the MissionMemory 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.safetycritical.StorageParameters;

import javax.safetycritical.annotate.SCJAllowed;
import javax.safetycritical.annotate.SCJRestricted;

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

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

3.6.15 MyLevel0Sequencer.java

```java
package samples.dynamiclevel0;

import javax.realtime.PriorityParameters;

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

import javax.safetycritical.annotate.SCJAllowed;
import javax.safetycritical.annotate.SCJRestricted;

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, StorageParameters s) {
    super(p, s);
  }

  @SCJAllowed(SUPPORT)
  @SCJRestricted({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

```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.safetycritical.StorageParameters;
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

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

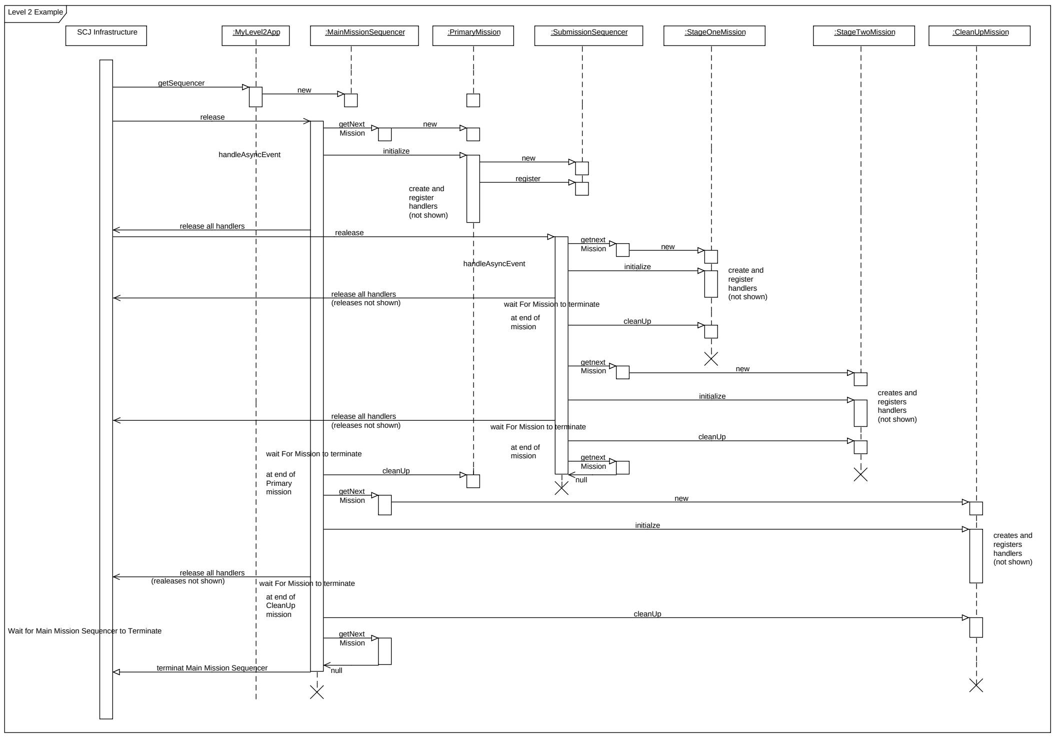
Level 2 Example
SCJ Infrastructure
:MyLevel2App
:MainMissionSequencer
:PrimaryMission
:SubmissionSequencer
:StageOneMission
:StageTwoMission
:CleanUpMission
getSequencer
new
release
getNext Mission
new
handleAsyncEvent
initialize
new
register
create and register handlers (not shown)
release all handlers
realease
getnext Mission
new
handleAsyncEvent
initialize
create and register handlers (not shown)
release all handlers (releases not shown)
wait For Mission to terminate
at end of mission
cleanUp
getnext Mission
new
initialize
creates and registers handlers (not shown)
release all handlers (releases not shown)
wait For Mission to terminate
cleanUp
at end of mission
getnext Mission
null
wait For Mission to terminate
at end of Primary mission
cleanUp
getNext Mission
new
initialze
creates and registers handlers (not shown)
release all handlers (realeases not shown)
wait For Mission to terminate
at end of CleanUp mission
cleanUp
Wait for Main Mission Sequencer to Terminate
getNext Mission
terminat Main Mission Sequencer
null

Figure 3.4: UML sequence diagram for Level 2 example

```
  @SCJAllowed(SUPPORT)
  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.safetycritical.Mission;
import javax.safetycritical.MissionSequencer;
import javax.safetycritical.Safelet;
import javax.safetycritical.StorageParameters;

import javax.safetycritical.annotate.SCJAllowed;

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() {
    StorageParameters sp =
      new StorageParameters(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.safetycritical.Mission;
import javax.safetycritical.MissionSequencer;
import javax.safetycritical.StorageParameters;

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,
                       StorageParameters 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.safetycritical.Mission;
import javax.safetycritical.StorageParameters;

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);
    StorageParameters sp =
      new StorageParameters(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.safetycritical.StorageParameters;

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);
    StorageParameters sp =
      new StorageParameters(100000L, null);
    MyCleanupThread t = new MyCleanupThread(pp, sp);
  }
}
```

3.6.23 SubMissionSequencer.java

```
package samples.level2;
```

```
import javax.realtime.PriorityParameters;

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

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,
                      StorageParameters 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.safetycritical.PeriodicEventHandler;
import javax.safetycritical.StorageParameters;

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 StorageParameters(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.safetycritical.ManagedThread;
import javax.safetycritical.StorageParameters;

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, StorageParameters 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

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 the 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 events are fired and an event handler executes in direct response to each fired 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 are more easily 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. For 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 the SCJ, it refers exclusively to either a bound asynchronous event handler or a no-heap real-time thread (called a ManagedThread in the SCJ).

An SCJ asynchronous event handler executes in response to each of a sequence of 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
- 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

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

[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 the same instruction sets.

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 and priority ceiling emulation is optional.

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 AffinitySets 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:

- Storage parameters – this permits, among other things, the storage used by a schedulable object's scoped memory area to be specified.
- Missions – all schedulable objects execute in the context of a mission (see Chapter 2).

4.1 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 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 to use Java, as well as to support the migration from traditional systems to more robust concurrent systems.

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.safetycritical.PriorityScheduler.instance().getMaxPriority() (that is, the maximum value for local ceilings – see Section 4.6.5).
- 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 and 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 the synchronized method) is not considered to be self-suspension.

4.2 Level Considerations

Specific semantics apply at each of the different compliance levels.

4.2.1 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 implementation 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.

4.2.2 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 (i.e., AperiodicEventHandler, AperiodicLongEventHandler and OneShotEventHandler) shall be supported.
- Each AperiodicEventHandler, AperiodicLongEventHandler, OneShotEventHandler or PeriodicEventHandler shall be permanently bound to its own implementation-defined thread of control, and each thread of control shall be bound to only a single 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. If application portability is a primary concern, the application should use no more than 28 priorities. There shall be no support 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. An implementation is required to document the time granularity at which missed deadlines are detected (see Section 4.7.5). The deadline miss shall be signalled no earlier than the deadline of the associated event handler.
- A preempted schedulable object shall be executed as it were placed at the front of the run queue for its active priority level. This is a recommendation in the RTSJ but is a requirement for SCJ.

4.2.3 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.
- 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.

4.3 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.

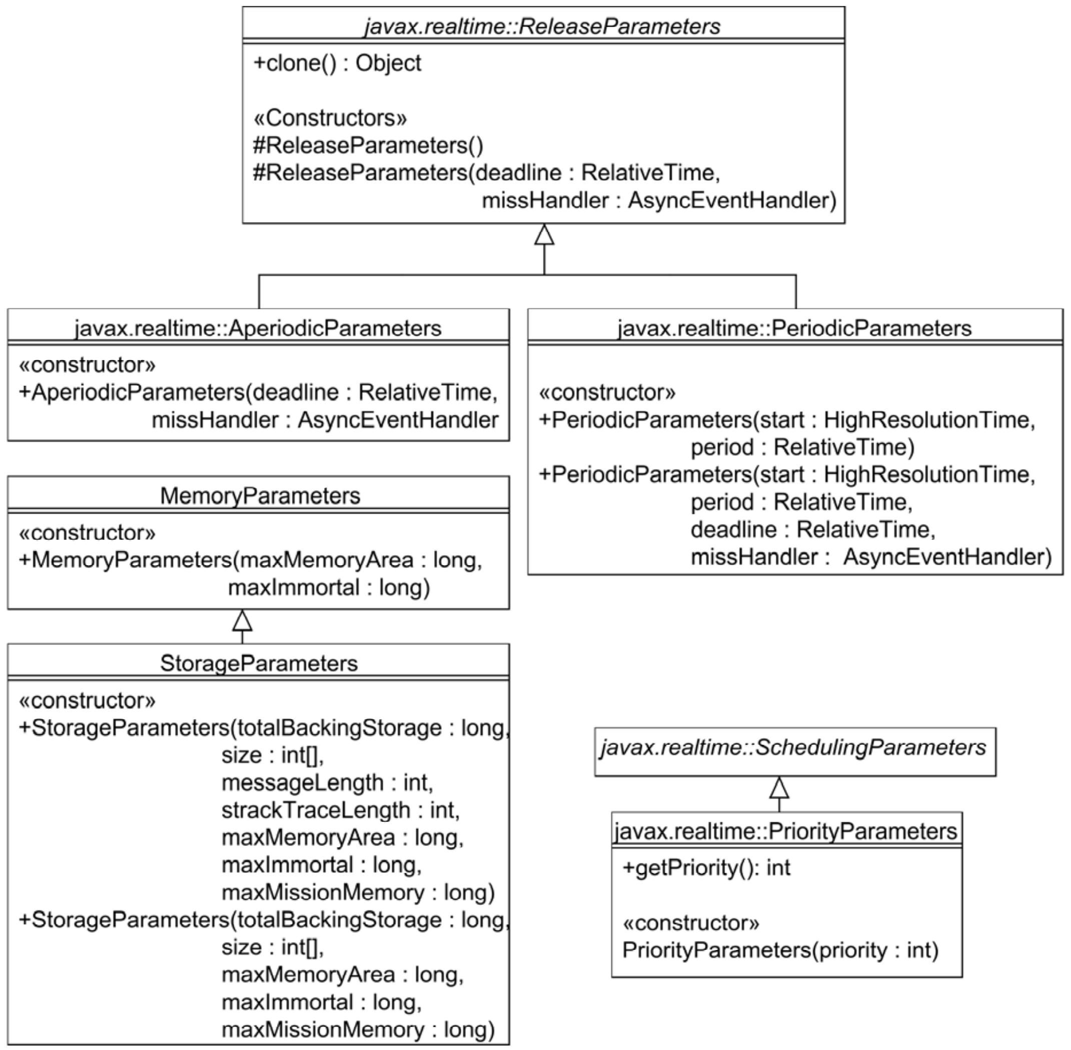
javax.realtime::ReleaseParameters
+clone() : Object
«Constructors»
#ReleaseParameters()
#ReleaseParameters(deadline : RelativeTime, missHandler : AsyncEventHandler)
javax.realtime::AperiodicParameters
«constructor»
+AperiodicParameters(deadline : RelativeTime, missHandler : AsyncEventHandler
javax.realtime::PeriodicParameters
«constructor»
+PeriodicParameters(start : HighResolutionTime, period : RelativeTime)
+PeriodicParameters(start : HighResolutionTime, period : RelativeTime, deadline : RelativeTime, missHandler : AsyncEventHandler)
MemoryParameters
«constructor»
+MemoryParameters(maxMemoryArea : long, maxImmortal : long)
StorageParameters
«constructor»
+StorageParameters(totalBackingStorage : long, size : int[], messageLength : int, strackTraceLength : int, maxMemoryArea : long, maxImmortal : long, maxMissionMemory : long)
+StorageParameters(totalBackingStorage : long, size : int[], maxMemoryArea : long, maxImmortal : long, maxMissionMemory : long)
javax.realtime::SchedulingParameters
javax.realtime::PriorityParameters
+getPriority(): int
«constructor»
PriorityParameters(priority : int)

Figure 4.1: Parameter classes

- 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 memory to be specified.

4.3.1 Class javax.realtime.ReleaseParameters

Declaration

@SCJAllowed
public abstract class ReleaseParameters
  implements java.lang.Cloneable
  extends java.lang.Object

Description

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

Constructors

@SCJAllowed
protected ReleaseParameters( )

Construct a ReleaseParameters object which has no deadline checking facility. There is no default for the deadline in this class. The default is set by the subclasses.

@SCJAllowed(javax.safetycritical.annotate.Level.LEVEL_1)
protected ReleaseParameters(RelativeTime deadline,
  AsyncEventHandler missHandler)

Construct an object which has deadline checking facility.

deadline — is a deadline to be checked.

missHandler — is the AsynchronousEventHandler to be released when the deadline miss has been detected.

Methods

@SCJAllowed
public java.lang.Object clone( )

Create a clone of this ReleaseParameters object.

4.3.2 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 @SCJRestricted(maySelfSuspend = false)
public PeriodicParameters(HighResolutionTime start, RelativeTime period)

Constructs a new PeriodicParameters object within the current memory area.

start — is the time of the first release of the associated schedulable object relative to the start of the mission. If the start time is in the past, the first release shall occur immediately. A null value defaults to an offset of zero milliseconds. An absolute start time is also measured relative to the start of the mission.

period — is the time between each release of the associated schedulable object. The default deadline is the same value as the period. The default handler is null.

Throws IllegalArgumentException if period is null.

@SCJAllowed @SCJRestricted(maySelfSuspend = false)
public PeriodicParameters(HighResolutionTime start,
RelativeTime period,
RelativeTime deadline,
AsyncEventHandler handler)

Construct a new PeriodicParameters object within the current memory area.

start — is time of the first release of the associated schedulable 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.

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

Throws IllegalArgumentException if period is null.

4.3.3 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)
public AperiodicParameters( )

Construct a new AperiodicParameters object within the current memory area with no deadline detection facility.

@SCJAllowed(javax.safetycritical.annotate.Level.LEVEL_1)
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.

4.3.4 Class javax.realtime.SchedulingParameters

Declaration

@SCJAllowed
public abstract class SchedulingParameters
  implements java.lang.Cloneable
  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.

4.3.5 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 4.6.5). 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 @SCJRestricted(maySelfSuspend = false)
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 @SCJRestricted(maySelfSuspend = false)
public int getPriority( )

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

4.3.6 Class javax.realtime.MemoryParameters

Declaration

@SCJAllowed
public class MemoryParameters
  implements java.lang.Cloneable
  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 NO_MAX

Constructors

@SCJAllowed
public MemoryParameters(**long** maxMemoryArea, **long** maxImmortal)

Create a MemoryParameters object with the given maximum values.

maxMemoryArea — 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 other than positive. zero, or NO_MAX is passed as the value of maxMemoryArea or maxImmortal.

4.3.7 Class javax.safetycritical.StorageParameters

Declaration

@SCJAllowed
public final class StorageParameters

  extends javax.realtime.MemoryParameters

Description

StorageParameters provide storage size parameters for ISRs and schedulable objects in a ManagedSchedulable: event handlers, threads, and sequencers. A StorageParameters object is passed as a parameter to the constructor of mission sequencers and other SCJ schedulable objects.

Constructors

```
@SCJAllowed
public StorageParameters(long totalBackingStore,
  long [] sizes,
  int messageLength,
  int stackTraceLength,
  long maxMemoryArea,
  long maxImmortal,
  long maxMissionMemory)
```

This is the primary constructor for a StorageParameters object, permitting specification of all settable values.

totalBackingStore — size of the backing store reservation for worst-case scope usage by the associated ManagedSchedulable: object, in bytes.

sizes — is an array of parameters for configuring VM resources such as native stack or Java stack size. The meanings of the entries in the array are vendor specific. The array passed is not stored in the object.

messageLength — memory space in bytes dedicated to the message associated with this ManagedSchedulable object's ThrowBoundaryError exception, plus references to the method names/identifiers in the stack backtrace.

stackTraceLength — is the number of elements in the StackTraceElement array dedicated to stack backtrace associated with this StorageParameters object's ThrowBoundaryError exception.

maxMemoryArea — 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.

maxMissionMemory — is the maximum amount of memory in the mission memory area required by the associated schedulable object.

Throws IllegalArgumentException if any value other than positive. zero, or NO_MAX is passed as the value of maxMemoryArea or maxImmortal.

```
@SCJAllowed
public StorageParameters(long totalBackingStore,
  long [] sizes,
long maxMemoryArea,
long maxImmortal,
long maxMissionMemory)
```

This is the secondary constructor for a StorageParameters object, permitting specification of backing size and an array of implementation-defined memory sizes.

totalBackingStore — size of the backing store reservation for worst-case scope usage in bytes.

sizes — is an array of parameters for configuring VM resources such as native stack or java stack size. The meaning of the entries in the array are vendor specific. The array passed in is not stored in the object.

maxMemoryArea — 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.

maxMissionMemory — is the maximum amount of memory in the mission memory area required by the associated schedulable object.

Throws IllegalArgumentException if any value other than positive. zero, or NO_MAX is passed as the value of maxMemoryArea or maxImmortal.

4.4 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.

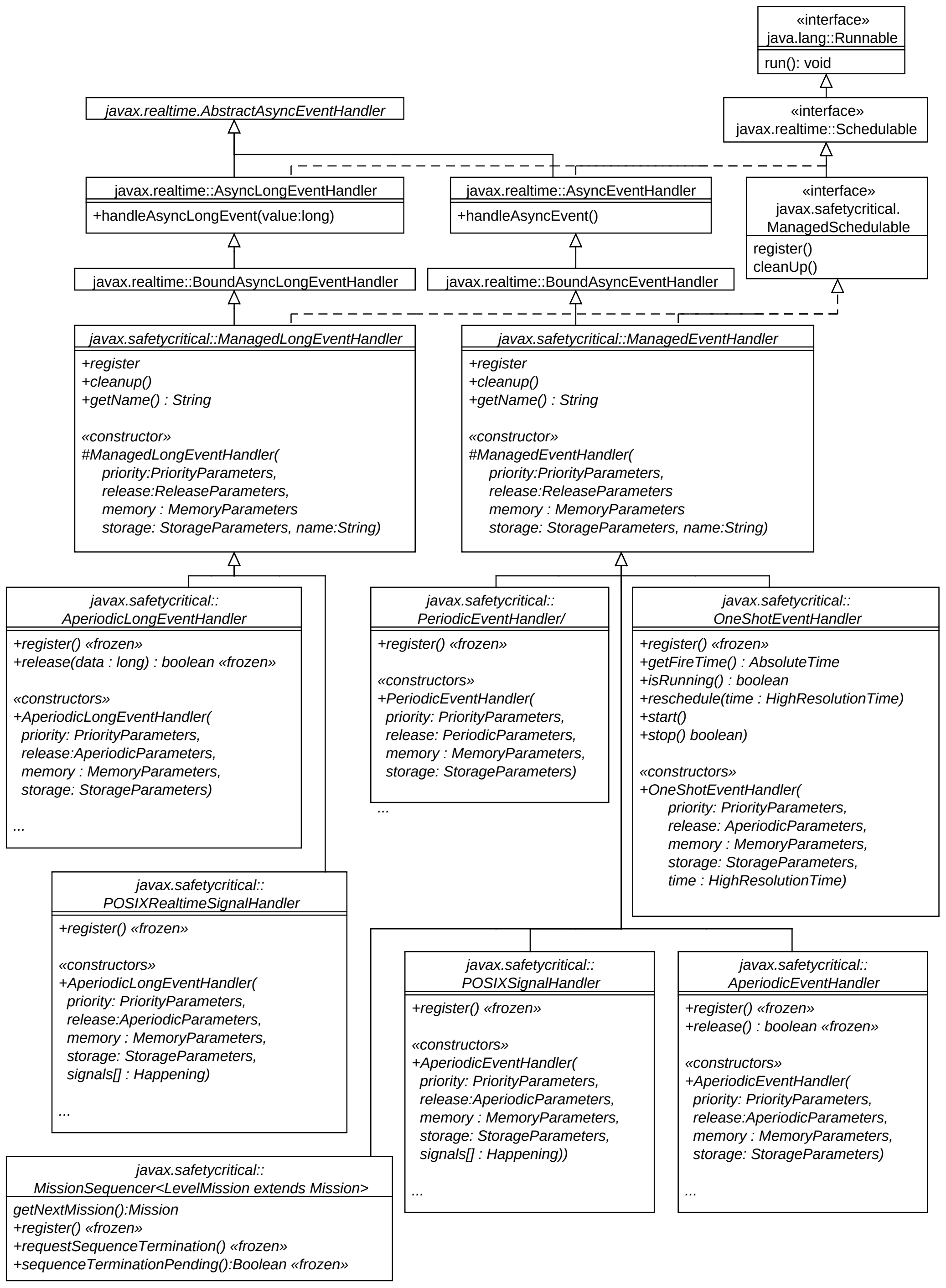
«interface»
java.lang::Runnable
run(): void
«interface»
javax.realtime::Schedulable
javax.realtime.AbstractAsyncEventHandler
javax.realtime::AsyncLongEventHandler
+handleAsyncLongEvent(value:long)
javax.realtime::AsyncEventHandler
+handleAsyncEvent()
«interface»
javax.safetycritical.
ManagedSchedulable
register()
cleanUp()
javax.realtime::BoundAsyncLongEventHandler
javax.realtime::BoundAsyncEventHandler
javax.safetycritical::ManagedLongEventHandler
+register
+cleanup()
+getName() : String
«constructor»
#ManagedLongEventHandler(
priority:PriorityParameters,
release:ReleaseParameters,
memory : MemoryParameters
storage: StorageParameters, name:String)
javax.safetycritical::ManagedEventHandler
+register
+cleanup()
+getName() : String
«constructor»
#ManagedEventHandler(
priority:PriorityParameters,
release:ReleaseParameters
memory : MemoryParameters
storage: StorageParameters, name:String)
javax.safetycritical::
AperiodicLongEventHandler
+register() «frozen»
+release(data : long) : boolean «frozen»
«constructors»
+AperiodicLongEventHandler(
priority: PriorityParameters,
release:AperiodicParameters,
memory : MemoryParameters,
storage: StorageParameters)
...
javax.safetycritical::
PeriodicEventHandler/
+register() «frozen»
«constructors»
+PeriodicEventHandler(
priority: PriorityParameters,
release: PeriodicParameters,
memory : MemoryParameters,
storage: StorageParameters)
...
javax.safetycritical::
OneShotEventHandler
+register() «frozen»
+getFireTime() : AbsoluteTime
+isRunning() : boolean
+reschedule(time : HighResolutionTime)
+start()
+stop() boolean)
«constructors»
+OneShotEventHandler(
priority: PriorityParameters,
release: AperiodicParameters,
memory : MemoryParameters,
storage: StorageParameters,
time : HighResolutionTime)
javax.safetycritical::
POSIXRealtimeSignalHandler
+register() «frozen»
«constructors»
+AperiodicLongEventHandler(
priority: PriorityParameters,
release:AperiodicParameters,
memory : MemoryParameters,
storage: StorageParameters,
signals[] : Happening)
...
javax.safetycritical::
POSIXSignalHandler
+register() «frozen»
«constructors»
+AperiodicEventHandler(
priority: PriorityParameters,
release:AperiodicParameters,
memory : MemoryParameters,
storage: StorageParameters,
signals[] : Happening))
...
javax.safetycritical::
AperiodicEventHandler
+register() «frozen»
+release() : boolean «frozen»
«constructors»
+AperiodicEventHandler(
priority: PriorityParameters,
release:AperiodicParameters,
memory : MemoryParameters,
storage: StorageParameters)
...
javax.safetycritical::
MissionSequencer<LevelMission extends Mission>
getNextMission():Mission
+register() «frozen»
+requestSequenceTermination() «frozen»
+sequenceTerminationPending():Boolean «frozen»

Figure 4.2: Handler classes

4.4.1 Class javax.realtime.Schedulable

Declaration

```
@SCJAllowed
public interface Schedulable implements java.lang.Runnable
```

Description

In keeping with the RTSJ, SCJ event handlers are schedulable objects. However, the Schedulable interface in the RTSJ is mainly concerned with on-line feasibility analysis and the getting and setting of the parameter classes. On the contrary, in SCJ, it provides no extra functionality over the Runnable interface.

4.4.2 Class javax.safetycritical.ManagedSchedulable

Declaration

```
@SCJAllowed
public interface ManagedSchedulable
  implements javax.realtime.Schedulable
```

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 applications classes.

Methods

```
@SCJAllowed(javax.safetycritical.annotate.Level.SUPPORT)
@SCJRestricted({javax.safetycritical.annotate.Phase.CLEANUP})
public void cleanUp( )
```

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

4.4.3 Class javax.realtime.AbstractAsyncEventHandler

Declaration

```
@SCJAllowed(javax.safetycritical.annotate.Level.LEVEL_0)
public abstract class AbstractAsyncEventHandler
  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.

4.4.4 Class javax.realtime.AsyncEventHandler

Declaration

@SCJAllowed(javax.safetycritical.annotate.Level.LEVEL_0)
public class AsyncEventHandler

extends javax.realtime.AbstractAsyncEventHandler

Description

In SCJ, all asynchronous events must 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)
public void handleAsyncEvent( )

This method must be overridden by the application to provide the handling code.

4.4.5 Class javax.realtime.AsyncLongEventHandler

Declaration

@SCJAllowed(javax.safetycritical.annotate.Level.LEVEL_0)
public class AsyncLongEventHandler

extends javax.realtime.AbstractAsyncEventHandler

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
public void handleAsyncEvent(**long** data)

This method must be overridden by the application to provide the handling code.

data — is the data that was passed when the associated event was fired.

4.4.6 Class javax.realtime.BoundAsyncEventHandler

Declaration

@SCJAllowed
public class BoundAsyncEventHandler

extends javax.realtime.AsyncEventHandler

Description

The BoundAsyncEventHandler class is not directly available to the safety-critical Java application developers. Hence none of its methods or constructors are publicly available.

4.4.7 Class javax.realtime.BoundAsyncLongEventHandler

Declaration

@SCJAllowed
public class BoundAsyncLongEventHandler

extends javax.realtime.AsyncLongEventHandler

Description

The BoundAsyncLongEventHandler class is not directly available to the safety-critical Java application developers. Hence none of its methods or constructors are publicly available. This class differs from BoundAsyncEventHandler in that when it is fired, a long integer is provided for use by the released event handler(s).

4.4.8 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.

Note that the values in parameter 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.

Methods

@SCJAllowed(javax.safetycritical.annotate.Level.SUPPORT)
@SCJRestricted({javax.safetycritical.annotate.Phase.CLEANUP})
public void cleanUp( )

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

MissionMemory is the current allocation context on entry into this method. When the cleanUp method is called, a private memory area shall be provided for its use, and shall be the current memory area. If desired, the cleanUp method may introduce a new PrivateMemory area. The memory allocated to ManagedSchedulables shall be available to be reclaimed when each Mission's cleanUp method returns.

@SCJAllowed
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.

4.4.9 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 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. They set up the initial memory area of each managed handler to be a private memory that is entered before a call to handleAsyncEvent and that is left on return. The size of the private memory area allocated is the maximum available to the infrastructure for this handler.

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.

This class differs from ManagedEventHandler in that when it is fired, a long integer is provided for use by the released event handler(s).

Constructors

```
@SCJAllowed
@SCJRestricted({javax.safetycritical.annotate.Phase.INITIALIZATION})
public ManagedLongEventHandler(PriorityParameters priority,
  ReleaseParameters release,
  StorageParameters storage,
  String name)
```

Constructs an event handler.

Does not perform memory allocation. Does not allow this to escape local scope. Builds links from this to priority, parameters, and name so those three arguments must reside in scopes that enclose this.

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 and period. 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. This argument must not be null.

storage — specifies the non-null maximum memory demands for this event handler.

Throws IllegalArgumentException IllegalArgumentException if priority, release or memory parameters are null.

Methods

```
@Override
@SCJAllowed(javax.safetycritical.annotate.Level.SUPPORT)
@SCJRestricted({javax.safetycritical.annotate.Phase.CLEANUP})
public void cleanUp( )
```

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

MissionMemory is the current allocation context on entry into this method.

```
@SCJAllowed
public java.lang.String getName( )
```

returns a string name for this handler, including its priority and its level.Registers this event handler with the current mission.

4.4.10 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.

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 4.7.4 for a discussion of the impact of this on application scheduling.

Constructors

@SCJAllowed
@SCJRestricted({javax.safetycritical.annotate.Phase.INITIALIZATION})
public PeriodicEventHandler(PriorityParameters priority,
PeriodicParameters release,
StorageParameters storage)

Constructs a periodic event handler.

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, rather it is relative to the point in time when initialization is finished and the timers are started. This argument must not be null.

storage — specifies the storage parameters for the periodic event handler. It must not be null.

Throws IllegalArgumentException IllegalArgumentException if priority, release or memory is null.

Memory behavior: Does not perform memory allocation. Does not allow this to escape local scope. Builds links from this to priority and parameters, so those two arguments must reside in scopes that enclose this.

```
@SCJAllowed(javax.safetycritical.annotate.Level.LEVEL_1)
@SCJRestricted({javax.safetycritical.annotate.Phase.INITIALIZATION})
public PeriodicEventHandler(PriorityParameters priority,
  PeriodicParameters release,
  StorageParameters storage,
  String name)
```

Constructs a periodic event handler.

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. This argument must not be null.

storage — specifies the memory parameters for the periodic event handler. It must not be null.

Throws IllegalArgumentException IllegalArgumentException if priority, release, or scp is null.

Memory behavior: Does not perform memory allocation. Does not allow this to escape local scope. Builds links from this to priority and parameters, so those two arguments must reside in scopes that enclose this.

Methods

```
@SCJAllowed(javax.safetycritical.annotate.Level.LEVEL_1)
public javax.realtime.HighResolutionTime getActualStartTime( )
```

Get the actual start time of this handler. The actual 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. If the actual start time is equal to the effect start time, then the method behaves as if getResquestStartTime() 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 a time parameter based on the clock used to start the timer.

@SCJAllowed(javax.safetycritical.annotate.Level.LEVEL_1)
public javax.realtime.HighResolutionTime getEffectiveStartTime( )

Get the effective start time of this handler. If the clock associated with the start time parameter and the interval 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 interval parameter (passed at construction time) when the handler is actually started.

returns a reference based on the clock associated with the interval parameter.

@SCJAllowed(javax.safetycritical.annotate.Level.LEVEL_1)
public javax.realtime.AbsoluteTime getLastReleaseTime( )

Get the last release time of this handler.

returns a reference to a newly-created **javax.safetycritical.AbsoluteTime** object representing this handlers's last release time, according to the clock associated with the intervale parameter used at construction time.

Throws IllegalStateException Thrown if this timer has not been released since it was last started.

@SCJAllowed(javax.safetycritical.annotate.Level.LEVEL_1)
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 **javax.safetycritical.AbsoluteTime** object. The clock association of the returned time is the clock on which interval parameter (passed at construction time) is based.

Throws ArithmeticException Thrown if the result does not fit in the normalized format.

Throws IllegalStateException Thrown if this handler has not been started.

@SCJAllowed(javax.safetycritical.annotate.Level.LEVEL_1)
public javax.realtime.HighResolutionTime 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 effect the requested start time of this handler if it has not already beend started.

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

@SCJAllowed(javax.safetycritical.annotate.Level.LEVEL_1)
public javax.realtime.AbsoluteTime getnextReleaseTime(AbsoluteTime dest)

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

dest — The instance of **javax.safetycritical.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 **javax.safetycritical.AbsoluteTime** passed as parameter, with time values representing 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 interval parameter (passed at construction time) is based.

Throws ArithmeticException Thrown if the result does not fit in the normalized format.

Throws IllegalStateException Thrown if this handler has not been started.

@SCJAllowed
@Override
@SCJRestricted({javax.safetycritical.annotate.Phase.INITIALIZATION})
public final void register( )

4.4.11 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 handle-AsyncEvent 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 4.7.4 for a discussion of the impact of this on application scheduling.

Constructors

```
@SCJAllowed
@SCJRestricted({javax.safetycritical.annotate.Phase.INITIALIZATION})
public OneShotEventHandler(PriorityParameters priority,
  HighResolutionTime time,
  AperiodicParameters release,
  StorageParameters memory)
```

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 release. 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. A null paramter is equivalent to a relative time of 0.

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

storage — specifies the storage parameters; it must not be null

Throws IllegalArgumentException IllegalArgumentException if priority, release or memory is null.

Memory behavior: Does not perform memory allocation. Does not allow this to escape local scope. Builds links from this to priority and parameters, so those two arguments must reside in scopes that enclose this.

```
@SCJAllowed(javax.safetycritical.annotate.Level.LEVEL_1)
@SCJRestricted({javax.safetycritical.annotate.Phase.INITIALIZATION})
public OneShotEventHandler(PriorityParameters priority,
  HighResolutionTime time,
  AperiodicParameters release,
  StorageParameters memory,
  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 release. 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. A null paramter is equivalent to a relative time of 0.

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

storage — specifies the storage parameters; it must not be null.

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

Throws IllegalArgumentException IllegalArgumentException if priority, release, or scp is null.

Memory behavior: Does not perform memory allocation. Does not allow this to escape local scope. Builds links from this to priority and parameters, so those two arguments must reside in scopes that enclose this.

Methods

@SCJAllowed(javax.safetycritical.annotate.Level.LEVEL_1)
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)
public javax.realtime.AbsoluteTime getNextReleaseTime(AbsoluteTime dest)

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

dest — The instance of **javax.safetycritical.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 **javax.safetycritical.AbsoluteTime** representing 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 interval parameter (passed at construction time) is based.

Throws IllegalStateException Thrown if this handler has not been started.

@SCJAllowed
@Override
@SCJRestricted({javax.safetycritical.annotate.Phase.INITIALIZATION})
public final void register( )

@SCJAllowed(javax.safetycritical.annotate.Level.LEVEL_1)
public void scheduleNextReleaseTime(HighResolutionTime time)

Change 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 released as if it was created using that type for its time parameter. An absolute time in the passed is equivalent to a relative time 0f (0,0). The rescheduling will take place between the invocation and the return of the method. <P> If scheduleNextReleaseTime is invoked if null </P>

Throws IllegalArgumentException Thrown if time is a negative RelativeTime value or null.

4.4.12 Class javax.safetycritical.AperiodicEventHandler

Declaration

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

extends javax.safetycritical.ManagedEventHandler

Description

This class permits the automatic execution of code that is bound to an aperiodic event. It is abstract. Concrete subclasses must implement the handleAsync-Event method and may override the default cleanup method.

Note, there is no programmer access to the RTSJ fireCount mechanisms, so the associated methods are missing.

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)
@SCJRestricted({javax.safetycritical.annotate.Phase.INITIALIZATION})
public AperiodicEventHandler(PriorityParameters priority,
  AperiodicParameters release,
  StorageParameters storage)

Constructs an aperiodic event handler that can be explicitly released.

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

release — specifies the release parameters for this aperiodic event handler; it must not be null.

stoarge — specifies the StorageParameters for this aperiodic event handler

Throws IllegalArgumentException IllegalArgumentException if priority, release or event is null.

Memory behavior: Does not perform memory allocation. Does not allow this to escape local scope. Builds links from this to priority and parameters, so those two arguments must reside in scopes that enclose this. Builds a link from this to the event, so the event must reside in memory that encloses this.

Methods

@SCJAllowed
public final void release( )

Release this aperiodic event handler

4.4.13 Class javax.safetycritical.AperiodicLongEventHandler

Declaration

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

extends javax.safetycritical.ManagedLongEventHandler

Description

This class permits the automatic execution of code that is bound to an aperiodic event. It is abstract. Concrete subclasses must implement the handleAsync-Event method and may override the default cleanup method.

Note, there is no programmer access to the RTSJ fireCount mechanisms, so the associated methods are missing.

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)
@SCJRestricted({javax.safetycritical.annotate.Phase.INITIALIZATION})
public AperiodicLongEventHandler(PriorityParameters priority,
  AperiodicParameters release,
  StorageParameters storage)
```

Constructs an aperiodic event handler that can be released.

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

release — specifies the release parameters for this aperiodic event handler; it must not be null.

storage — specifies the storage parameters for the periodic event handler. It must not be null.

Throws IllegalArgumentException IllegalArgumentException if priority, release or event is null.

Memory behavior: Does not perform memory allocation. Does not allow this to escape local scope. Builds links from this to priority and parameters, so those two arguments must reside in scopes that enclose this. Builds a link from this to event, so event must reside in memory that encloses this.

Methods

```
@SCJAllowed
public final void release(long data)
```

Release this aperiodic event handler

4.5 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.

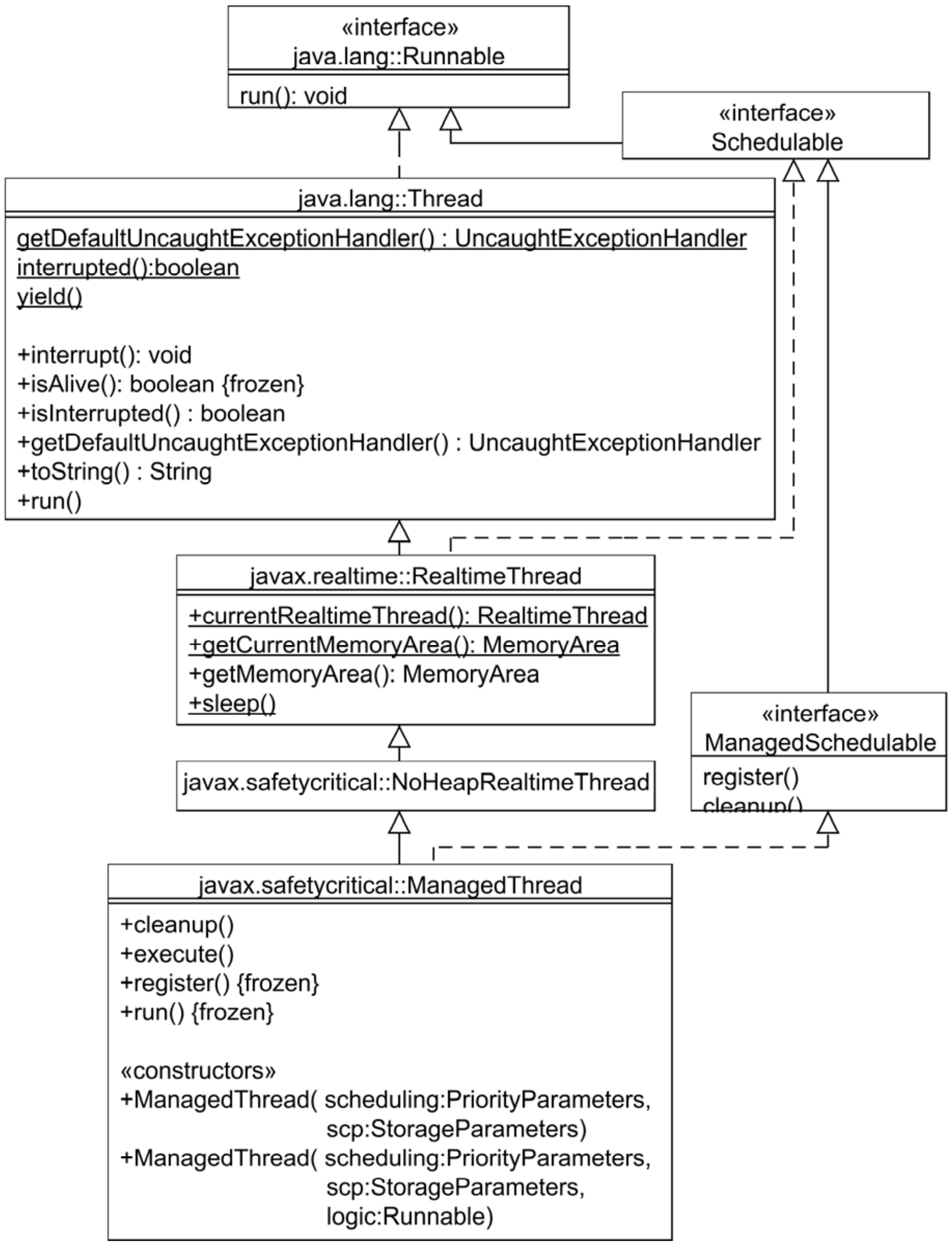
«interface»
java.lang::Runnable
run(): void
«interface»
Schedulable
java.lang::Thread
getDefaultUncaughtExceptionHandler() : UncaughtExceptionHandler
interrupted():boolean
yield()
+interrupt(): void
+isAlive(): boolean {frozen}
+isInterrupted() : boolean
+getDefaultUncaughtExceptionHandler() : UncaughtExceptionHandler
+toString() : String
+run()
javax.realtime::RealtimeThread
+currentRealtimeThread(): RealtimeThread
+getCurrentMemoryArea(): MemoryArea
+getMemoryArea(): MemoryArea
+sleep()
«interface»
ManagedSchedulable
register()
cleanup()
javax.safetycritical::NoHeapRealtimeThread
javax.safetycritical::ManagedThread
+cleanup()
+execute()
+register() {frozen}
+run() {frozen}
«constructors»
+ManagedThread(scheduling:PriorityParameters, scp:StorageParameters)
+ManagedThread(scheduling:PriorityParameters, scp:StorageParameters, logic:Runnable)

Figure 4.3: Thread classes

4.5.1 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 the 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)
@SCJRestricted({javax.safetycritical.annotate.Phase.ALL},
   maySelfSuspend = false,
   mayAllocate = true)
public static
java.lang.Thread.UncaughtExceptionHandler getDefaultUncaughtExceptionHandler( )
```

returns the default handler for uncaught exceptions.

Memory behavior: 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)
@SCJRestricted({javax.safetycritical.annotate.Phase.ALL},
   maySelfSuspend = false,
   mayAllocate = true)
public
java.lang.Thread.UncaughtExceptionHandler getUncaughtExceptionHandler( )
```

returns the handler invoked when this thread abruptly terminates due to an uncaught exception.

Memory behavior: 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)
```

```
@SCJRestricted({javax.safetycritical.annotate.Phase.ALL},
   maySelfSuspend = false,
   mayAllocate = true)
```
public void interrupt( )

Interrupts this thread.

Memory behavior: Allocates no memory. Does not allow this to escape local variables.

```
@SCJAllowed(javax.safetycritical.annotate.Level.LEVEL_2)
@SCJRestricted(maySelfSuspend = false)
```
public static boolean interrupted( )

Tests whether the current thread has been interrupted. The interrupted status of the thread is cleared by this method.

returns true if the current thread has been interrupted.

Memory behavior: Allocates no memory. Does not allow this to escape local variables.

```
@SCJAllowed(javax.safetycritical.annotate.Level.LEVEL_2)
@SCJRestricted({javax.safetycritical.annotate.Phase.ALL},
   maySelfSuspend = false,
   mayAllocate = true)
```
public final boolean isAlive( )

returns true if the current thread has not returned from run().

Memory behavior: Allocates no memory. Does not allow this to escape local variables.

```
@SCJAllowed(javax.safetycritical.annotate.Level.LEVEL_2)
@SCJRestricted({javax.safetycritical.annotate.Phase.ALL},
   maySelfSuspend = false,
   mayAllocate = true)
```
public boolean isInterrupted( )

Tests whether this thread has been interrupted. The interrupted status of the thread is not affected by this method.

returns true if the current thread has been interrupted.

Memory behavior: Allocates no memory. Does not allow this to escape local variables.

@SCJAllowed(javax.safetycritical.annotate.Level.SUPPORT)
public void run( )

> Wait for completion of the thread, but do not wait longer than millis milliseconds and nanos nanoseconds. This method can be overridden to provide the code of the thread at Level 2. @memory Does not allow this to escape local variables.

@SCJRestricted({javax.safetycritical.annotate.Phase.ALL},
    maySelfSuspend = false,
    mayAllocate = true)
@SCJAllowed(javax.safetycritical.annotate.Level.LEVEL_2)
public static void setDefaultUncaughtExceptionHandler(
  Thread.UncaughtExceptionHandler eh)

> 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.

eh — is the default handler to be set.

Memory behavior: Allocates no memory. Does not allow this to escape local variables. The eh argument must reside in immortal memory.

Memory behavior: This constructor requires that the "eh" argument reside in a scope that encloses the scope of the "@immortal" argument.

@SCJAllowed(javax.safetycritical.annotate.Level.LEVEL_2)
@SCJRestricted({javax.safetycritical.annotate.Phase.ALL},
    maySelfSuspend = false,
    mayAllocate = false)
public void setUncaughtExceptionHandler(
  Thread.UncaughtExceptionHandler eh)

eh — the UncaughtExceptionHandler to be set for this thread.

Memory behavior: Allocates no memory. Does not allow this to escape local variables. The eh argument must reside in a scope that encloses the scope of this.

Memory behavior: This constructor requires that the "eh" argument reside in a scope that encloses the scope of the "this" argument.

```
@SCJRestricted({javax.safetycritical.annotate.Phase.ALL},
   maySelfSuspend = false,
   mayAllocate = true)
@SCJAllowed
public java.lang.String toString( )
```

returns a string representation of this thread, including the thread's name and priority.

Memory behavior: Does not allow this to escape local variables. Allocates a String and associated internal "structure" (e.g. char[]) in caller's scope.

Memory behavior: This constructor may allocate objects within the currently active MemoryArea.

```
@SCJAllowed(javax.safetycritical.annotate.Level.LEVEL_2)
@SCJRestricted({javax.safetycritical.annotate.Phase.ALL},
   maySelfSuspend = true,
   mayAllocate = false)
public static void yield( )
```

Memory behavior: Allocates no memory. Causes the currently executing thread object to temporary pause and allow other threads to execute.

4.5.2 Class java.lang.Thread.UncaughtExceptionHandler

Declaration

@SCJAllowed
public interface UncaughtExceptionHandler

Description

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
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.

4.5.3 Class javax.realtime.RealtimeThread

Declaration

@SCJAllowed(javax.safetycritical.annotate.Level.LEVEL_1)
public class RealtimeThread
  implements javax.realtime.Schedulable
  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 getCurrentMemoryArea method can be used at Level 1, hence the class is visible at Level 1.

Methods

@SCJAllowed(javax.safetycritical.annotate.Level.LEVEL_1)
@SCJRestricted(maySelfSuspend = false, mayAllocate = false)
public static javax.realtime.MemoryArea getCurrentMemoryArea( )

Allocates no memory. The returned object may reside in scoped memory, within a scope that encloses the current execution context.

returns a reference to the current allocation context.

@SCJAllowed(javax.safetycritical.annotate.Level.LEVEL_2)
@SCJRestricted(maySelfSuspend = false, mayAllocate = false)
public javax.realtime.MemoryArea getMemoryArea( )

Allocates no memory. Does not allow this to escape local variables. The returned object may reside in scoped memory, within a scope that encloses this.

returns a reference to the initial allocation context represented by this.

@SCJAllowed(javax.safetycritical.annotate.Level.LEVEL_2)
@SCJRestricted(maySelfSuspend = true)
public static void sleep(HighResolutionTime time)
throws java.lang.InterruptedException

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

Throws java.lang.IllegalArgumentException if time is based on a user-defined clock that does not drive events.

4.5.4 Class javax.realtime.NoHeapRealtimeThread

Declaration

@SCJAllowed(javax.safetycritical.annotate.Level.LEVEL_2)
public class NoHeapRealtimeThread **extends** javax.realtime.RealtimeThread

Description

NoHeapRealtimeThreads} cannot be directly created by the SCJ application. However, they are needed by the infrastructure to support ManagedThreads at Level 2.

Constructors

@SCJAllowed(javax.safetycritical.annotate.Level.LEVEL_2)
public NoHeapRealtimeThread(SchedulingParameters schedule,
MemoryArea area)

TBD: do we use this constructor, which expects a MemoryArea argument?

```
@SCJAllowed(javax.safetycritical.annotate.Level.LEVEL_2)
public NoHeapRealtimeThread(SchedulingParameters schedule,
  ReleaseParameters release)
```

TBD: do we use this constructor, which expects a ReleaseParameters argument?

Methods

```
@SCJAllowed(javax.safetycritical.annotate.Level.LEVEL_2)
@SCJRestricted(mayAllocate = false, maySelfSuspend = false)
public void start( )
```

Creation of thread may block, but start shall not block.

4.5.5 Class javax.safetycritical.ManagedThread

Declaration

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

Description

This class enables a mission to keep track of all the no-heap realtime threads that are created during the initialization phase. It also sets up the initial memory area for the thread to be a private memory, whose size is the maximum that the infrastructure can assign to this thread.

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

```
@SCJAllowed(javax.safetycritical.annotate.Level.LEVEL_2)
@SCJRestricted({javax.safetycritical.annotate.Phase.INITIALIZATION})
public ManagedThread(SchedulingParameters priority,
  StorageParameters storage)
```

Constructs a thread that is managed by the enclosing mission.

Does not allow this to escape local variables. Creates a link from the constructed object to the scheduling, storage, and logic parameters. Thus, all of these parameters must reside in a scope that encloses this.

The priority represented by scheduling parameter is consulted only once, at construction time. If scheduling.getPriority() returns different values at different times, only the initial value is honored.

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

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

Throws IllegalArgumentException if priority or storage is null.

Memory behavior: This constructor requires that the "priority" argument reside in a scope that encloses the scope of the "this" argument. This constructor requires that the "storage" argument reside in a scope that encloses the scope of the "this" argument.

```
@SCJAllowed(javax.safetycritical.annotate.Level.LEVEL_2)
@SCJRestricted({javax.safetycritical.annotate.Phase.INITIALIZATION})
public ManagedThread(PriorityParameters priority,
  StorageParameters storage,
  Runnable logic)
```

Creates a thread that is managed by the enclosing mission.

Does not allow this to escape local variables. Creates a link from the constructed object to the priority, memory, and logic parameters . Thus, all of these parameters must reside in a scope that encloses this.

The priority represented by priority parameter is consulted only once, at construction time. If priority.getPriority() returns different values at different times, only the initial value is honored.

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.

Memory behavior: This constructor requires that the "priority" argument reside in a scope that encloses the scope of the "this" argument. This constructor requires that the "mem_info" argument reside in a scope that encloses the scope of the "this" argument. This constructor requires that the "logic" argument reside in a scope that encloses the scope of the "this" argument.

Methods

@Override
@SCJAllowed(javax.safetycritical.annotate.Level.LEVEL_2)
@SCJRestricted({})
public void cleanUp( )

Execute any clean up code associated with this managed thread. This method is called by the infrastructure, so that it is not callable from any phase. In fact, it encapsulates the entire cleanup phase.

@Override
@SCJAllowed(javax.safetycritical.annotate.Level.LEVEL_0)
@SCJRestricted({javax.safetycritical.annotate.Phase.INITIALIZATION})
public final void register( )

Register this managed thread. Note: method made Level 0 since it overrides ManagedSchedulable.register() method which is also Level 0. Note, however, that this does not make the method visible at Level 0 since the enclosing class is Level 2.

4.6 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 preemptive priority scheduler.

4.6.1 Class javax.safetycritical.CyclicSchedule

See Section 3.4.5.

4.6.2 Class javax.safetycritical.CyclicExecutive

See Section 3.4.6.

4.6.3 Class javax.realtime.Scheduler

Declaration

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

Description

The RTSJ supports generic on-line feasibility analysis via the Scheduler class. SCJ supports off-line analysis; hence most of the methods in this class are omitted. Only the static method getCurrentSO is provided.

Methods

@SCJAllowed
public static javax.realtime.Schedulable getCurrentSO( )

returns the current asynchronous event handler or real-time thread of the caller.

4.6.4 Class javax.realtime.PriorityScheduler

Declaration

@SCJAllowed(javax.safetycritical.annotate.Level.LEVEL_1)
public 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(javax.safetycritical.annotate.Level.LEVEL_1)
@SCJRestricted(maySelfSuspend = false)
public int getMaxPriority( )

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

@SCJAllowed(javax.safetycritical.annotate.Level.LEVEL_1)
@SCJRestricted(maySelfSuspend = false)
public int getMinPriority( )

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

4.6.5 Class javax.safetycritical.PriorityScheduler

Declaration

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

Description

The SCJ priority scheduler supports the notion of both software and hardware priorities.

Methods

@SCJAllowed(javax.safetycritical.annotate.Level.LEVEL_1)
@SCJRestricted(maySelfSuspend = false)
public int getMaxHardwarePriority( )

returns the maximum hardware real-time priority supported by this scheduler.

@SCJAllowed(javax.safetycritical.annotate.Level.LEVEL_1)
@SCJRestricted(maySelfSuspend = false)
public int getMinHardwarePriority( )

returns the minimum hardware real-time priority supported by this scheduler.

@SCJAllowed(javax.safetycritical.annotate.Level.LEVEL_1)
public static javax.safetycritical.PriorityScheduler instance( )

returns the SCJ priority scheduler

4.6.6 Class javax.realtime.AffinitySet

Declaration

@SCJAllowed
public final class AffinitySet **extends** java.lang.Object

Description

This class is the API for all processor-affinity-related aspects of SCJ. It includes a factory that generates AffinitySet objects, and methods that control the default affinity sets used when affinity set inheritance does not apply.

Affinity sets implement the concept of SCJ scheduling allocation domains. They provide the mechanism by which the programmer can specify the processors on which managed schedulable objects can execute.

The processor membership of an affinity set is immutable. SCJ constrains the use of RTSJ affinity sets so that the affinity of a managed schedulable object can only be set during the initialization phase.

The internal representation of a set of processors in an affinity set instance is not specified. Each processor/core in the system is given a unique logical number. The relationship between logical and physical processors is implementation-defined.

The affinity set factory cannot create an affinity set with more than one processor member, but such affinity sets are supported as pre-defined affinity sets at Level 2.

A managed schedulable object inherits its creator's affinity set. Every managed schedulable object is associated with a processor affinity set instance, either explicitly assigned, inherited, or defaulted.

See also Services.getSchedulingAllocationDoamins()

Methods

@SCJAllowed(javax.safetycritical.annotate.Level.LEVEL_2)
public static javax.realtime.AffinitySet generate(**int** processorNumber)

Generates an affinity set consisting of a single processor.

returns An AffinitySet representing a single processors in the system. The returned object may be dynamically created in the current memory area or preallocated in immortal memory.

Throws IllegalArgumentException if processorNumber is not a valid processor in the set of processors allocated to the JVM.

@SCJAllowed(javax.safetycritical.annotate.Level.LEVEL_2)
public static final javax.realtime.AffinitySet getAffinitySet(
Thread thread)

returns an AffinitySet representing the set of processors on which thread can be scheduled. The returned object may be dynamically created in the current memory area or preallocated in immortal memory.

Throws throw NullPointerException if thread is null.

@SCJAllowed(javax.safetycritical.annotate.Level.LEVEL_1)
public static final javax.realtime.AffinitySet getAffinitySet(
BoundAsyncEventHandler handler)

returns an AffinitySet representing the set of processors on which handler can be scheduled. The returned object may be dynamically created in the current memory area or preallocated in immortal memory.

Throws NullPointerException if handler is null.

@SCJAllowed(javax.safetycritical.annotate.Level.LEVEL_2)
public final boolean isProcessorInSet(**int** processorNumber)

true if and only if the processorNumber is in this affinity set.

@SCJAllowed(javax.safetycritical.annotate.Level.LEVEL_2)
public static final void setProcessorAffinity(AffinitySet set,
Thread thread)

Set the set of processors on which thread can be scheduled to that represented by set}.

set — is the required affinity set

thread — is the target managed thread.

Throws ProcessorAffinityException if set is not a valid processor set, and NullPointerException if thread is null

@SCJAllowed(javax.safetycritical.annotate.Level.LEVEL_1)
public static final void setProcessorAffinity(AffinitySet set,
BoundAsyncEventHandler aeh)

Set the set of processors on which aeh can be scheduled to that represented by set.

set — is the required affinity set

aeh — is the taget bound async event handler

Throws ProcessorAffinityException if set is not a valid processor set, and NullPointerException if handler is null

4.6.7 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
public static void captureBackTrace(Throwable association)

Captures the stack back trace for the current thread into its thread-local stack back trace buffer and remembers that the current contents of the stack back trace buffer is associated with the object represented by the association argument. The size of the stack back trace buffer is determined by the StorageParameters object that is passed as an argument to the constructor of the corresponding Schedulable. If the stack back trace buffer is not large enough to capture all of the stack back trace information, the information is truncated in an implementation-defined manner.

@SCJAllowed
public static
javax.safetycritical.ManagedSchedulable currentManagedSchedulable( )

@returns
a reference to the currently executed ManagedEventHandler or ManagedThread.

@SCJAllowed(javax.safetycritical.annotate.Level.LEVEL_2)
@SCJRestricted(maySelfSuspend = true)
public static void delay(HighResolutionTime delay)

This is like sleep except that it is not interruptible and it uses nanoseconds instead of milliseconds.

delay — is the number of nanoseconds to suspend. if delay is a RelativeTime type then it represents the number of milliseconds and nanoseconds to suspend. If delay is a time in the past, the method returns immediately.

Throws IllegalArgumentException if the clock associated with delay does not drive events.

@SCJAllowed(javax.safetycritical.annotate.Level.LEVEL_2)
@SCJRestricted(maySelfSuspend = true)
public static void delay(**int** ns_delay)

This is like sleep except that it is not interruptible and it uses nanoseconds instead of milliseconds.

ns_delay — is the number of nanoseconds to suspend

@SCJAllowed(javax.safetycritical.annotate.Level.LEVEL_1)
public static int getDefaultCeiling( )

returns the default ceiling priority. The default ceiling priority is the PriorityScheduler.getMaxPriority. It is assumed that this can be changed using a virtual machine configuration option.

@SCJAllowed
public static javax.safetycritical.annotate.Level getDeploymentLevel( )

returns the deployment compliance level

@SCJAllowed(javax.safetycritical.annotate.Level.LEVEL_0)
@SCJRestricted(maySelfSuspend = false)
public static
javax.realtime.AffinitySet[] getSchedulingAllocationDoamins( )

@SCJAllowed(javax.safetycritical.annotate.Level.LEVEL_0)
@SCJRestricted(maySelfSuspend = false)
public static void nanoSpin(**int** nanos)

Busy wait in nano seconds.

nanos —

@SCJAllowed(javax.safetycritical.annotate.Level.LEVEL_1)
@SCJRestricted({javax.safetycritical.annotate.Phase.INITIALIZATION})
public static void setCeiling(Object O, **int** pri)

Sets the ceiling priority of object O The priority pri can be in the software or hardware priority range. Ceiling priorities are immutable.

Throws IllegalThreadStateException if called outside the mission phase

@SCJAllowed(javax.safetycritical.annotate.Level.LEVEL_0)
@SCJRestricted(maySelfSuspend = false)
public static void spin(HighResolutionTime delay)

Busy wait spinning loop (until now plus delay).

delay — If delay is a RelativeTime type then it represents the number of milliseconds and nanoseconds to suspend. If delay is a time in the past, the method returns immediately.

4.7 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, they may be, and generally will be, including 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 that 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 the most recent version of the language standard (Ada 2005) includes an explicit subset of Ada tasking constructs, called the *Ravenscar Profile*, that are amenable to formal certification against standards such as DO-178B.

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.

4.7.1 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 is 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. As the priority ceiling emulation protocol is optional in the RTSJ and compulsory in SCJ, 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. This means that a synchronized method can only call a method that will not self-suspend.

4.7.2 Multiprocessors

Although the techniques for analyzing the timing properties of multiprocessor systems are still relatively in their infancy, there is general acceptance 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.

4.7.3 Feasibility Analysis and MultiProcessors

While 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 feasibility analysis is equivalent to the well-known single processor feasibility analysis, but would be carried out for each scheduling allocation domain.

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 feasibility 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.

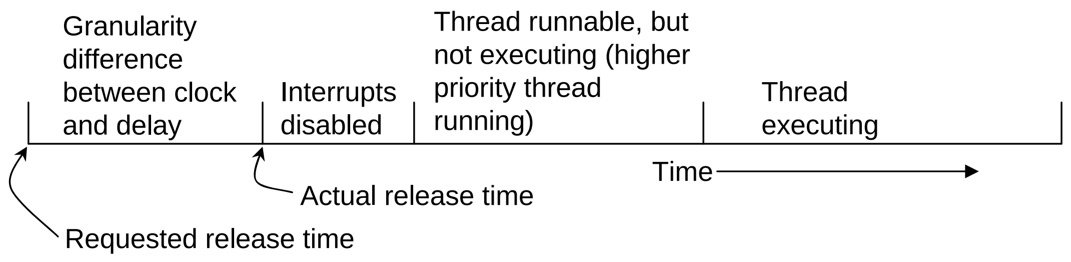
Granularity difference between clock and delay
Interrupts disabled
Thread runnable, but not executing (higher priority thread running)
Thread executing
Time
Actual release time
Requested release time

Figure 4.4: Granularity of delays

4.7.4 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 getResolution 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.

4.7.5 Deadline Miss Detection

Although SCJ supports deadline miss detection, it is important to understand the intrinsic limitations of the facility. The SCJ supports deadline overrun detection only for event handlers at levels 1 and 2. As explained in Section 4.7.4, all time-triggered computation 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.

With a periodic event handler, it is possible that the clock associated with the deadline is distinct from the clock that releases the event handler that is to be monitored by the deadline detection. Whenever deadlines are associated with different clocks than the clock that releases the monitored event handler, the sequence of events is as illustrated in Figure 4.5. Note that monitoring of the deadline cannot begin until the event handler begins to run. Once it does begin to run, it is necessary to wait for 1 more tick on the deadline overrun clock than the minimum number of ticks that spans the deadline's intended time span, just in case the deadline monitoring begins immediately before delivery of the first tick. Once the deadline overrun handler becomes eligible to run, its actual execution will depend on its priority relative to other schedulable objects.

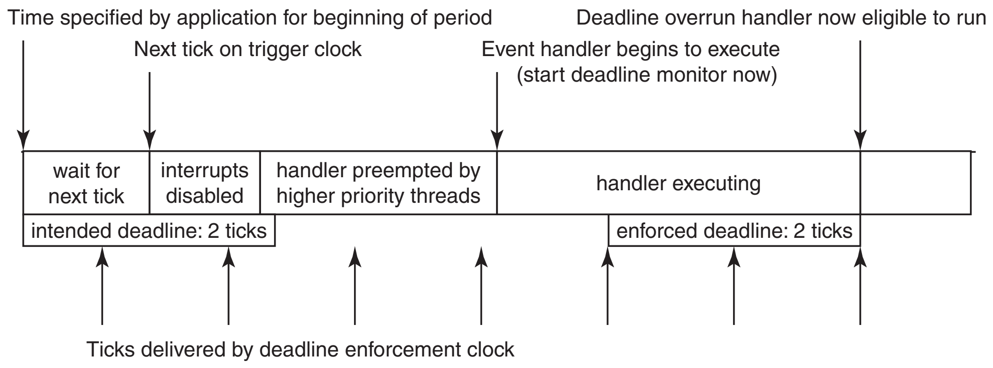
Time specified by application for beginning of period
Deadline overrun handler now eligible to run
Next tick on trigger clock
Event handler begins to execute (start deadline monitor now)
wait for next tick
interrupts disabled
handler preempted by higher priority threads
handler executing
intended deadline: 2 ticks
enforced deadline: 2 ticks
Ticks delivered by deadline enforcement clock

Figure 4.5: Imprecision in deadline enforcement for time-triggered event on different clock than deadline clock

In the case that a deadline is enforced on a time-triggered event handler and the deadline is expressed relative to the same clock that caused release of the monitored event handler, the timing relationships are as illustrated in Figure 4.6. Because deadline enforcement is based on the same clock as the event handler's release, the infrastructure is able to align the start of the deadline monitor with release of the event.

In both cases, if a deadline is not an integral multiple of the deadline enforcement clock, the infrastructure must round the requested deadline up to the nearest multiple of the tick size.

A related limitation is that a deadline can be missed but not detected. This can occur when the deadline has been set at a smaller granularity than the detecting timer. 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 $D + \Delta$, i will have missed its deadline, but this miss will have been undetected.

A third 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

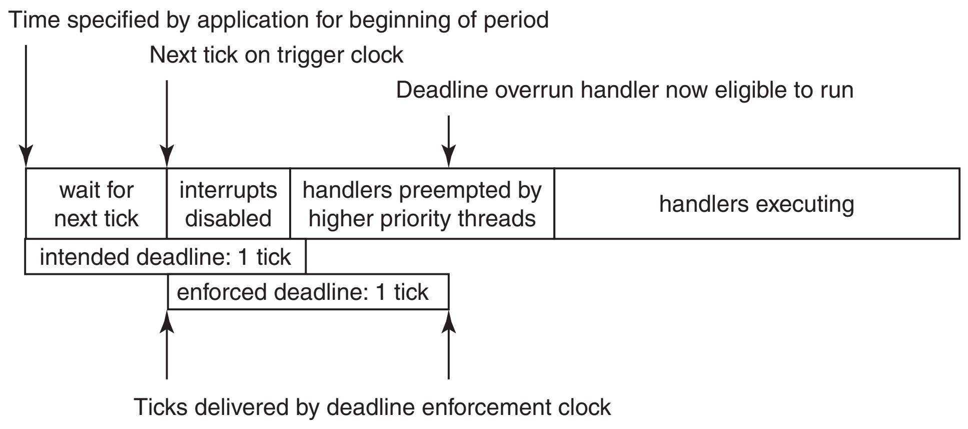
Time specified by application for beginning of period
Next tick on trigger clock
Deadline overrun handler now eligible to run
wait for next tick
interrupts disabled
handlers preempted by higher priority threads
handlers executing
intended deadline: 1 tick
enforced deadline: 1 tick
Ticks delivered by deadline enforcement clock

Figure 4.6: Imprecision in deadline enforcement for time-triggered event on same clock as deadline clock

event that signals the miss of a deadline. This is clearly a race condition. The timer event could fire between the last statement of the handleAsyncEvent method and the canceling of the timer event. Hence a deadline miss could be signalled when arguably the application had performed all of its computation.

4.8 Compatibility

The following incompatibilities exist with RTSJ Version 1.1.

- PCE is the default monitor control policy in SCJ whereas priority inheritance is the default in the RTSJ
- There are no corresponding facilities in the RTSJ to facilitate the specification of the SCJStorageParameters.
- There is no support for a separate range of hardware priorities in the RTSJ.

Chapter 5

Interaction with Devices and External Events

This chapter presents the facilities provided by SCJ to aid in the interaction between the application and the external environment.

This interaction can be partitioned between:

- accessing external input/output devices and handling interrupts, and
- handling operating system signals.

In general, interfacing to input/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. These facilities are specified in section 5.1. SCJ supports optional first level interrupt handling when this can be provided by the underlying execution environment. These optional features are defined in the InterruptServiceRoutine class specified in section 5.2.

Handling of operating signals is undertaken is specified in section **??**.

5.1 Raw Memory Access

5.1.1 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 or memory into which DMA can be performed. 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* and *removable memory*. The following specifies the SCJ's facilities for raw memory access:

- Each type of raw memory access is identified by a tagging interface called RawMemoryName.
- The raw memory name MEM_ACCESS facilitates access to memory locations that are outside the main memory used by the JVM. This may be, for example, memory that is shared with other processes.
- The raw memory name IO_PORT_MAPPED 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 raw memory name IO_MEMORY_MAPPED 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 name DMA_ACCESS facilitates access to memory locations that are outside the main memory used by the JVM. It is used in conjunction with devices which allow DMA transfers.
- Access to raw memory is enforced by implementation-defined objects, called **accessor objects**. These objects implement specification-defined interfaces (e.g., RawByte, RawByteArray etc) and are created by implementation-defined factory objects. Each factory implements the RawIntegralAccessFactory interface, and is identified by its raw memory name.
- Only Java integral types are supported.
- The RawMemory 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 general memory access.

2. The created factory is then registered with the raw memory manager.

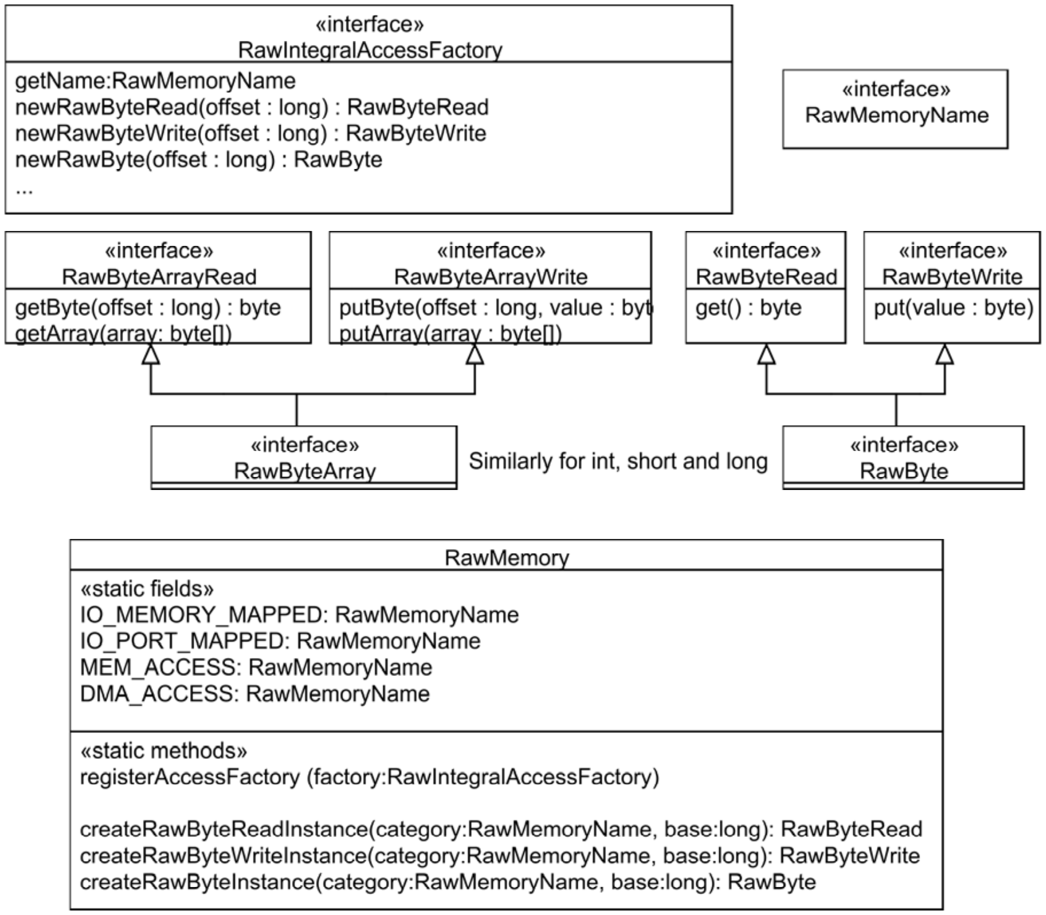
«interface»
RawIntegralAccessFactory
getName:RawMemoryName
newRawByteRead(offset : long) : RawByteRead
newRawByteWrite(offset : long) : RawByteWrite
newRawByte(offset : long) : RawByte
...
«interface»
RawMemoryName
«interface»
RawByteArrayRead
getByte(offset : long) : byte
getArray(array: byte[])
«interface»
RawByteArrayWrite
putByte(offset : long, value : byt
putArray(array : byte[])
«interface»
RawByteRead
get() : byte
«interface»
RawByteWrite
put(value : byte)
«interface»
RawByteArray
Similarly for int, short and long
«interface»
RawByte
RawMemory
«static fields»
IO_MEMORY_MAPPED: RawMemoryName
IO_PORT_MAPPED: RawMemoryName
MEM_ACCESS: RawMemoryName
DMA_ACCESS: RawMemoryName
«static methods»
registerAccessFactory (factory:RawIntegralAccessFactory)
createRawByteReadInstance(category:RawMemoryName, base:long): RawByteRead
createRawByteWriteInstance(category:RawMemoryName, base:long): RawByteWrite
createRawByteInstance(category:RawMemoryName, base:long): RawByte

Figure 5.1: Raw memory classes and interfaces

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 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.

5.1.2 Level Considerations

The defined facilities are available at all compliance levels.

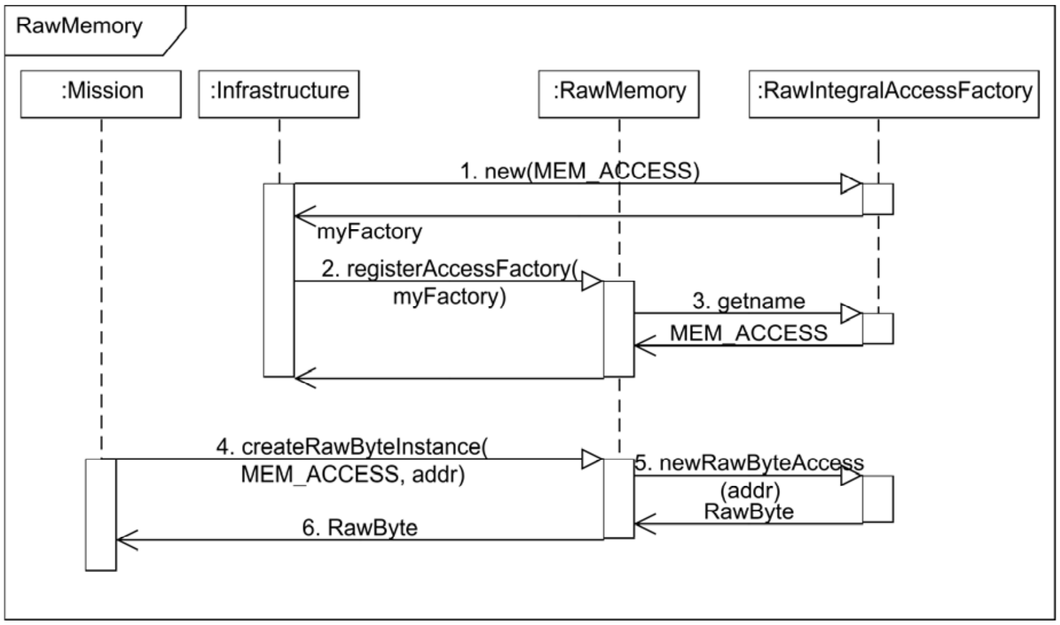

Figure 5.2: Raw memory classes interactions

5.1.3 javax.realtime.RawMemoryName

Declaration

@SCJAllowed(javax.safetycritical.annotate.Level.LEVEL_0)
public interface RawMemoryName

Description

A tagging interface for marking objects that identify raw memory types.

5.1.4 javax.realtime.RawByte

Declaration

@SCJAllowed(javax.safetycritical.annotate.Level.LEVEL_0)
public interface RawByte
implements javax.realtime.RawByteRead, javax.realtime.RawByteWrite

Description

An interface to a byte accessor object. An accessor object encapsules the protocol required to access a byte in raw memory.

5.1.5 javax.realtime.RawByteRead

Declaration

@SCJAllowed(javax.safetycritical.annotate.Level.LEVEL_0)
public interface RawByteRead

Description

An interface to a byte accessor object. An accessor object encapsules the protocol required to read a byte in raw memory.

Methods

@SCJAllowed(javax.safetycritical.annotate.Level.LEVEL_0)
@SCJRestricted(mayAllocate = false, maySelfSuspend = false)
public byte get( )

Get the value of this raw byte.

returns the byte from raw memory.

5.1.6 javax.realtime.RawByteWrite

Declaration

@SCJAllowed(javax.safetycritical.annotate.Level.LEVEL_0)
public interface RawByteWrite

Description

An interface to a byte accessor object. An accessor object encapsules the protocol required to write a byte in raw memory.

Methods

@SCJAllowed(javax.safetycritical.annotate.Level.LEVEL_0)
@SCJRestricted(mayAllocate = false, maySelfSuspend = false)
public void put(**byte** value)

Store the byte value in the associated raw memory.

value — is the byte to be stored.

5.1.7 javax.realtime.RawByteArrayRead

Declaration

@SCJAllowed(javax.safetycritical.annotate.Level.LEVEL_0)
public interface RawByteArrayRead

Description

An interface to a byte array read accessor object. An accessor object encapsules the protocol required to read a byte array in raw memory.

Methods

@SCJAllowed(javax.safetycritical.annotate.Level.LEVEL_0)
@SCJRestricted(mayAllocate = false, maySelfSuspend = false)
public byte get(**long** offset)

Get the value from this raw byte array.

returns the byte from the array in raw memory.

@SCJAllowed(javax.safetycritical.annotate.Level.LEVEL_0)
@SCJRestricted(mayAllocate = false, maySelfSuspend = false)
public void get(**byte** [] array)

Get the value of this raw byte array into array.

5.1.8 javax.realtime.RawByteArrayWrite

Declaration

@SCJAllowed(javax.safetycritical.annotate.Level.LEVEL_0)
public interface RawByteArrayWrite

Description

An interface to a byte array write accessor object. An accessor object encapsules the protocol required to write a byte array in raw memory.

Methods

@SCJAllowed(javax.safetycritical.annotate.Level.LEVEL_0)
@SCJRestricted(mayAllocate = false, maySelfSuspend = false)
public void put(**byte** b, **long** offset)

Store the byte in array in the associated Raw memory.

b — is the byte to be stored.

offset — is the location of the byte to be stored relative to the raw memory area.

@SCJAllowed(javax.safetycritical.annotate.Level.LEVEL_0)
@SCJRestricted(mayAllocate = false, maySelfSuspend = false)
public void put(**byte** [] i)

Store the byte array value in the associated Raw memory.

i — is the array to be stored.

5.1.9 javax.realtime.RawByteArray

Declaration

@SCJAllowed(javax.safetycritical.annotate.Level.LEVEL_0)
public interface RawByteArray
implements javax.realtime.RawByteArrayRead, javax.realtime.RawByteArrayWrite

Description

An interface to a byte array accessor object. An accessor object encapsules the protocol required to access a byte array in raw memory.

5.1.10 javax.realtime.RawShort

Declaration

@SCJAllowed(javax.safetycritical.annotate.Level.LEVEL_0)
public interface RawShort
implements javax.realtime.RawShortRead, javax.realtime.RawShortWrite

Description

An interface to a short accessor object. An accessor object encapsulates the protocol required to access a short in raw memory.

5.1.11 javax.realtime.RawShortRead

Declaration

@SCJAllowed(javax.safetycritical.annotate.Level.LEVEL_0)
public interface RawShortRead

Description

An interface to a short accessor object. An accessor object encapsulates the protocol required to read a short in raw memory.

Methods

@SCJAllowed(javax.safetycritical.annotate.Level.LEVEL_0)
@SCJRestricted(mayAllocate = false,
maySelfSuspend = false,
{javax.safetycritical.annotate.Phase.ALL})
public short get( )

Get the value of this raw short.

returns the short from raw memory.

5.1.12 javax.realtime.RawShortWrite

Declaration

@SCJAllowed(javax.safetycritical.annotate.Level.LEVEL_0)
public interface RawShortWrite

Description

An interface to a short accessor object. An accessor object encapsulates the protocol required to write a short in raw memory.

Methods

@SCJAllowed(javax.safetycritical.annotate.Level.LEVEL_0)
@SCJRestricted(mayAllocate = false,
  maySelfSuspend = false,
  {javax.safetycritical.annotate.Phase.ALL})
public void set(**short** value)

Store the short value in the associated Raw memory.

value — is the short to be stored.

5.1.13 javax.realtime.RawShortArrayRead

Declaration

@SCJAllowed(javax.safetycritical.annotate.Level.LEVEL_0)
public interface RawShortArrayRead

Description

An interface to a short array accessor object. An accessor object encapsules the protocol required to access a short array in raw memory.

Methods

@SCJAllowed(javax.safetycritical.annotate.Level.LEVEL_0)
@SCJRestricted(mayAllocate = false,
  maySelfSuspend = false,
  {javax.safetycritical.annotate.Phase.ALL})
public short get(**long** offset)

Get the value of a short from this raw short array.

returns the short from raw memory.

```
@SCJAllowed(javax.safetycritical.annotate.Level.LEVEL_0)
@SCJRestricted(mayAllocate = false,
   maySelfSuspend = false,
   {javax.safetycritical.annotate.Phase.ALL})
public void get(short [] array)
```

Get the value of this raw short array into array.

array — is the array to place the data.

5.1.14 javax.realtime.RawShortArrayWrite

Declaration

```
@SCJAllowed(javax.safetycritical.annotate.Level.LEVEL_0)
public interface RawShortArrayWrite
```

Description

An interface to a short array accessor object for writing. An accessor object encapsules the protocol required to write access a short array in raw memory.

Methods

```
@SCJAllowed(javax.safetycritical.annotate.Level.LEVEL_0)
@SCJRestricted(mayAllocate = false,
   maySelfSuspend = false,
   {javax.safetycritical.annotate.Phase.ALL})
public void put(short value, long offset)
```

Store the short in the associated Raw memory array.

value — is the short to be stored.

offset — is the position in array to be stored.

```
@SCJAllowed(javax.safetycritical.annotate.Level.LEVEL_0)
@SCJRestricted(mayAllocate = false,
   maySelfSuspend = false,
   {javax.safetycritical.annotate.Phase.ALL})
public void put(short [] array)
```

Store the short array value in the associated Raw memory.

array — is the array to be stored.

5.1.15 javax.realtime.RawShortArray

Declaration

@SCJAllowed(javax.safetycritical.annotate.Level.LEVEL_0)
public interface RawShortArray
implements javax.realtime.RawShortArrayRead, javax.realtime.RawShortArrayWrite

Description

An interface to a short array accessor object. An accessor object encapsules the protocol required to access a short array in raw memory.

5.1.16 javax.realtime.RawInt

Declaration

@SCJAllowed(javax.safetycritical.annotate.Level.LEVEL_0)
public interface RawInt
implements javax.realtime.RawIntRead, javax.realtime.RawIntWrite

Description

An interface to a int accessor object. An accessor object encapsulates the protocol required to access a int in raw memory.

5.1.17 javax.realtime.RawIntRead

Declaration

@SCJAllowed(javax.safetycritical.annotate.Level.LEVEL_0)
public interface RawIntRead

Description

An interface to a int accessor object. An accessor object encapsulates the protocol required to read a int in raw memory.

Methods

@SCJAllowed(javax.safetycritical.annotate.Level.LEVEL_0)
@SCJRestricted(mayAllocate = false, maySelfSuspend = false)
public int get( )

Get the value of this raw int.

returns the int from raw memory.

5.1.18 javax.realtime.RawIntWrite

Declaration

@SCJAllowed(javax.safetycritical.annotate.Level.LEVEL_0)
public interface RawIntWrite

Description

An interface to a int accessor object. An accessor object encapsulates the protocol required to write a int in raw memory.

Methods

@SCJAllowed(javax.safetycritical.annotate.Level.LEVEL_0)
@SCJRestricted(mayAllocate = false, maySelfSuspend = false)
public void put(**int** value)

Store the int value in the associated Raw memory.

value — is the value to be stored.

5.1.19 javax.realtime.RawIntArrayRead

Declaration

@SCJAllowed(javax.safetycritical.annotate.Level.LEVEL_0)
public interface RawIntArrayRead

Description

An interface to a int array accessor object. An accessor object encapsules the protocol required to access a int array in raw memory.

Methods

@SCJAllowed(javax.safetycritical.annotate.Level.LEVEL_0)
@SCJRestricted(mayAllocate = false, maySelfSuspend = false)
public int get(**long** offset)

Get the value of a int from this raw int array.

returns the int from raw memory.

@SCJAllowed(javax.safetycritical.annotate.Level.LEVEL_0)
@SCJRestricted(mayAllocate = false, maySelfSuspend = false)
public void get(**int** [] array)

Get the value of this raw int array into array.

array — is the array to place the data.

5.1.20 javax.realtime.RawIntArrayWrite

Declaration

@SCJAllowed(javax.safetycritical.annotate.Level.LEVEL_0)
public interface RawIntArrayWrite

Description

An interface to a int array accessor object for writing. An accessor object encapsules the protocol required to write access a int array in raw memory.

Methods

@SCJAllowed(javax.safetycritical.annotate.Level.LEVEL_0)
@SCJRestricted(mayAllocate = false, maySelfSuspend = false)
public void put(**int** value, **long** offset)

Store the int in the associated Raw memory array.

value — is the int to be stored.

offset — is the position in array to be stored.

@SCJAllowed(javax.safetycritical.annotate.Level.LEVEL_0)
@SCJRestricted(mayAllocate = false, maySelfSuspend = false)
public void put(**int** [] array)

Store the int array value in the associated Raw memory.

array — is the array to be stored.

5.1.21 javax.realtime.RawIntArray

Declaration

@SCJAllowed(javax.safetycritical.annotate.Level.LEVEL_0)
public interface RawIntArray
  implements javax.realtime.RawIntArrayRead, javax.realtime.RawIntArrayWrite

Description

An interface to a int array accessor object. An accessor object encapsules the protocol required to access a int array in raw memory.

5.1.22 javax.realtime.RawLong

Declaration

@SCJAllowed(javax.safetycritical.annotate.Level.LEVEL_0)
public interface RawLong
implements javax.realtime.RawLongRead, javax.realtime.RawLongWrite

Description

An interface to a long accessor object. An accessor object encapsulates the protocol required to access a long in raw memory.

5.1.23 javax.realtime.RawLongRead

Declaration

@SCJAllowed(javax.safetycritical.annotate.Level.LEVEL_0)
public interface RawLongRead

Description

An interface to a long accessor object. An accessor object encapsulates the protocol required to read a long in raw memory.

Methods

@SCJAllowed(javax.safetycritical.annotate.Level.LEVEL_0)
@SCJRestricted(mayAllocate = false, maySelfSuspend = false)
public long get( )

Get the value of this raw long.

returns the long from raw memory.

5.1.24 javax.realtime.RawLongWrite

Declaration

@SCJAllowed(javax.safetycritical.annotate.Level.LEVEL_0)
public interface RawLongWrite

Description

An interface to a long accessor object. An accessor object encapsulates the protocol required to write a long in raw memory.

Methods

@SCJAllowed(javax.safetycritical.annotate.Level.LEVEL_0)
@SCJRestricted(mayAllocate = false, maySelfSuspend = false)
public void put(**long** value)

Store the long value in the associated Raw memory.

value — is the value to be stored.

5.1.25 javax.realtime.RawLongArrayRead

Declaration

@SCJAllowed(javax.safetycritical.annotate.Level.LEVEL_0)
public interface RawLongArrayRead

Description

An longerface to a long array accessor object. An accessor object encapsules the protocol required to access a long array in raw memory.

Methods

@SCJAllowed(javax.safetycritical.annotate.Level.LEVEL_0)
@SCJRestricted(mayAllocate = false, maySelfSuspend = false)
public long get(**long** offset)

Get the value of a long from this raw long array.

returns the long from raw memory.

@SCJAllowed(javax.safetycritical.annotate.Level.LEVEL_0)
@SCJRestricted(mayAllocate = false, maySelfSuspend = false)
public void get(**long** [] array)

Get the value of this raw long array longo array.

array — is the array to place the data.

5.1.26 javax.realtime.RawLongArrayWrite

Declaration

@SCJAllowed(javax.safetycritical.annotate.Level.LEVEL_0)
public interface RawLongArrayWrite

Description

An longerface to a long array accessor object for writing. An accessor object encapsules the protocol required to write access a long array in raw memory.

Methods

@SCJAllowed(javax.safetycritical.annotate.Level.LEVEL_0)
@SCJRestricted(mayAllocate = false, maySelfSuspend = false)
public void put(**long** value, **long** offset)

Store the long in the associated Raw memory array.

value — is the long to be stored.

offset — is the position in array to be stored.

@SCJAllowed(javax.safetycritical.annotate.Level.LEVEL_0)
@SCJRestricted(mayAllocate = false, maySelfSuspend = false)
public void put(**long** [] array)

Store the long array value in the associated Raw memory.

array — is the array to be stored.

5.1.27 javax.realtime.RawLongArray

Declaration

@SCJAllowed(javax.safetycritical.annotate.Level.LEVEL_0)
public interface RawLongArray
implements javax.realtime.RawLongArrayRead, javax.realtime.RawLongArrayWrite

Description

An interface to a long array accessor object. An accessor object encapsules the protocol required to access a long array in raw memory.

5.1.28 javax.realtime.RawIntegralAccessFactory

Declaration

@SCJAllowed(javax.safetycritical.annotate.Level.LEVEL_0)
public interface RawIntegralAccessFactory

Description

An interface that describes factory classes that create the accessor objects for raw memory access.

Methods

```
@SCJAllowed(javax.safetycritical.annotate.Level.LEVEL_0)
@SCJRestricted(mayAllocate = false, maySelfSuspend = false)
public javax.realtime.RawMemoryName getName( )
```

returns a reference to an object that implements the RawMemoryName interface. This “name” is associated with this factory and indirectly with all the objects created by this factory.

```
@SCJAllowed(javax.safetycritical.annotate.Level.LEVEL_0)
@SCJRestricted({javax.safetycritical.annotate.Phase.
   INTERRUPT_SERVICE_ROUTINE}, mayAllocate = false, maySelfSuspend = false)
public javax.realtime.RawByte newRawByte(long offset)
```

Creates an accessor object for accessing a byte in raw memory.

returns an object implementing the RawByte interface.

Throws AlignmentError if the offset is not on the appropriate boundary.

Throws SizeOutOfBoundsException if the byte falls in an invalid address range.

Throws MemoryTypeConflictException if offset does not point to memory that matches the type served by this factory.

Throws OffsetOutOfBoundsException if the offset is negative or greater than the size of the raw memory area.

```
@SCJAllowed(javax.safetycritical.annotate.Level.LEVEL_0)
@SCJRestricted({javax.safetycritical.annotate.Phase.
   INTERRUPT_SERVICE_ROUTINE}, mayAllocate = false, maySelfSuspend = false)
public javax.realtime.RawByteArray newRawByteArray(long base,
  int entries)
```

Creates an accessor object for accessing a byte array in raw memory.

returns an object implementing the RawByteArray interface.

Throws AlignmentError if the base is not on the appropriate boundary.

Throws SizeOutOfBoundsException if the byte array falls in an invalid address range.

Throws MemoryTypeConflictException if base does not point to memory that matches the type served by this factory.

Throws OffsetOutOfBoundsException if the base is negative or greater than the size of the raw memory area.

```
@SCJAllowed(javax.safetycritical.annotate.Level.LEVEL_0)
@SCJRestricted({javax.safetycritical.annotate.Phase.
   INTERRUPT_SERVICE_ROUTINE}, mayAllocate = false, maySelfSuspend = false)
public javax.realtime.RawByteArrayRead newRawByteArrayRead(long base,
  int entries)
```

Creates an accessor object for read accessing a byte array in raw memory.

returns an object implementing the RawByteArray interface.

Throws AlignmentError if the base is not on the appropriate boundary.

Throws SizeOutOfBoundsException if the byte array falls in an invalid address range.

Throws MemoryTypeConflictException if base does not point to memory that matches the type served by this factory.

Throws OffsetOutOfBoundsException if the base is negative or greater than the size of the raw memory area.

```
@SCJAllowed(javax.safetycritical.annotate.Level.LEVEL_0)
@SCJRestricted({javax.safetycritical.annotate.Phase.
   INTERRUPT_SERVICE_ROUTINE}, mayAllocate = false, maySelfSuspend = false)
public javax.realtime.RawByteArrayWrite newRawByteArrayWrite(
  long base,
  int entries)
```

Creates an accessor object for write accessing a byte array in raw memory.

returns an object implementing the RawByteArray interface.

Throws AlignmentError if the base is not on the appropriate boundary.

Throws SizeOutOfBoundsException if the byte array falls in an invalid address range.

Throws MemoryTypeConflictException if base does not point to memory that matches the type served by this factory.

Throws OffsetOutOfBoundsException if the base is negative or greater than the size of the raw memory area.

```
@SCJAllowed(javax.safetycritical.annotate.Level.LEVEL_0)
@SCJRestricted({javax.safetycritical.annotate.Phase.
   INTERRUPT_SERVICE_ROUTINE}, mayAllocate = false, maySelfSuspend = false)
public javax.realtime.RawByteRead newRawByteRead(long offset)
```

Creates an accessor object for read accessing a byte in raw memory.

returns an object implementing the RawByte interface.

Throws AlignmentError if the offset is not on the appropriate boundary.

Throws SizeOutOfBoundsException if the byte falls in an invalid address range.

Throws MemoryTypeConflictException if offset does not point to memory that matches the type served by this factory.

Throws OffsetOutOfBoundsException if the offset is negative or greater than the size of the raw memory area.

```
@SCJAllowed(javax.safetycritical.annotate.Level.LEVEL_0)
@SCJRestricted({javax.safetycritical.annotate.Phase.
   INTERRUPT_SERVICE_ROUTINE}, mayAllocate = false, maySelfSuspend = false)
public javax.realtime.RawByteWrite newRawByteWrite(long offset)
```

Creates an accessor object for write accessing a byte in raw memory.

returns an object implementing the RawByte interface.

Throws AlignmentError if the offset is not on the appropriate boundary.

Throws SizeOutOfBoundsException if the byte falls in an invalid address range.

Throws MemoryTypeConflictException if offset does not point to memory that matches the type served by this factory.

Throws OffsetOutOfBoundsException if the offset is negative or greater than the size of the raw memory area.

```
@SCJAllowed(javax.safetycritical.annotate.Level.LEVEL_0)
@SCJRestricted({javax.safetycritical.annotate.Phase.
   INTERRUPT_SERVICE_ROUTINE}, mayAllocate = false, maySelfSuspend = false)
public javax.realtime.RawInt newRawInt(long offset)
```

Creates an accessor object for accessing an int in raw memory.

returns an object implementing the RawInt interface.

Throws AlignmentError if the offset is not on the appropriate boundary.

Throws SizeOutOfBoundsException if the int falls in an invalid address range.

Throws OffsetOutOfBoundsException if the offset is negative or greater than the size of the raw memory area.

Throws MemoryTypeConflictException if offset does not point to memory that matches the type served by this factory.

@SCJAllowed(javax.safetycritical.annotate.Level.LEVEL_0)
@SCJRestricted({javax.safetycritical.annotate.Phase.
INTERRUPT_SERVICE_ROUTINE}, mayAllocate = false, maySelfSuspend = false)
public javax.realtime.RawIntArray newRawIntArray(**long** base, **int** entries)

Creates an accessor object for accessing a int array in raw memory.

returns an object implementing the RawIntArray interface.

Throws AlignmentError if the base is not on the appropriate boundary.

Throws SizeOutOfBoundsException if the int array falls in an invalid address range.

Throws MemoryTypeConflictException if base does not point to memory that matches the type served by this factory.

Throws OffsetOutOfBoundsException if the base is negative or greater than the size of the raw memory area.

@SCJAllowed(javax.safetycritical.annotate.Level.LEVEL_0)
@SCJRestricted({javax.safetycritical.annotate.Phase.
INTERRUPT_SERVICE_ROUTINE}, mayAllocate = false, maySelfSuspend = false)
public javax.realtime.RawIntArrayRead newRawIntArrayRead(**long** base,
int entries)

Creates an accessor object for read accessing a int array in raw memory.

returns an object implementing the RawIntArray interface.

Throws AlignmentError if the base is not on the appropriate boundary.

Throws SizeOutOfBoundsException if the int array falls in an invalid address range.

Throws MemoryTypeConflictException if base does not point to memory that matches the type served by this factory.

Throws OffsetOutOfBoundsException if the base is negative or greater than the size of the raw memory area.

@SCJAllowed(javax.safetycritical.annotate.Level.LEVEL_0)
@SCJRestricted({javax.safetycritical.annotate.Phase.
INTERRUPT_SERVICE_ROUTINE}, mayAllocate = false, maySelfSuspend = false)
public javax.realtime.RawIntArrayWrite newRawIntArrayWrite(**long** base,
int entries)

Creates an accessor object for write accessing a int array in raw memory.

returns an object implementing the RawIntArray interface.

Throws AlignmentError if the base is not on the appropriate boundary.

Throws SizeOutOfBoundsException if the int array falls in an invalid address range.

Throws MemoryTypeConflictException if base does not point to memory that matches the type served by this factory.

Throws OffsetOutOfBoundsException if the base is negative or greater than the size of the raw memory area.

```
@SCJAllowed(javax.safetycritical.annotate.Level.LEVEL_0)
@SCJRestricted({javax.safetycritical.annotate.Phase.
   INTERRUPT_SERVICE_ROUTINE}, mayAllocate = false, maySelfSuspend = false)
public javax.realtime.RawIntRead newRawIntRead(long offset)
```

Creates an accessor object for read accessing an int in raw memory.

returns an object implementing the RawInt interface.

Throws AlignmentError if the offset is not on the appropriate boundary.

Throws SizeOutOfBoundsException if the int falls in an invalid address range.

Throws OffsetOutOfBoundsException if the offset is negative or greater than the size of the raw memory area.

Throws MemoryTypeConflictException if offset does not point to memory that matches the type served by this factory.

```
@SCJAllowed(javax.safetycritical.annotate.Level.LEVEL_0)
@SCJRestricted({javax.safetycritical.annotate.Phase.
   INTERRUPT_SERVICE_ROUTINE}, mayAllocate = false, maySelfSuspend = false)
public javax.realtime.RawIntWrite newRawIntWrite(long offset)
```

Creates an accessor object for write accessing an int in raw memory.

returns an object implementing the RawInt interface.

Throws AlignmentError if the offset is not on the appropriate boundary.

Throws SizeOutOfBoundsException if the int falls in an invalid address range.

Throws OffsetOutOfBoundsException if the offset is negative or greater than the size of the raw memory area.

Throws MemoryTypeConflictException if offset does not point to memory that matches the type served by this factory.

```
@SCJAllowed(javax.safetycritical.annotate.Level.LEVEL_0)
@SCJRestricted({javax.safetycritical.annotate.Phase.
   INTERRUPT_SERVICE_ROUTINE}, mayAllocate = false, maySelfSuspend = false)
public javax.realtime.RawLong newRawLong(long offset)
```

Creates an accessor object for accessing a long in raw memory.

returns an object implementing the RawLong interface.

Throws AlignmentError if the offset is not on the appropriate boundary.

Throws SizeOutOfBoundsException if the long falls in an invalid address range.

Throws OffsetOutOfBoundsException if the offset is negative or greater than the size of the raw memory area.

Throws MemoryTypeConflictException if offset does not point to memory that matches the type served by this factory.

```
@SCJAllowed(javax.safetycritical.annotate.Level.LEVEL_0)
@SCJRestricted({javax.safetycritical.annotate.Phase.
   INTERRUPT_SERVICE_ROUTINE}, mayAllocate = false, maySelfSuspend = false)
public javax.realtime.RawLongArray newRawLongArray(long base,
  int entries)
```

Creates an accessor object for accessing a long array in raw memory.

returns an object implementing the RawLongArray interface.

Throws AlignmentError if the base is not on the appropriate boundary.

Throws SizeOutOfBoundsException if the long array falls in an invalid address range.

Throws MemoryTypeConflictException if base does not point to memory that matches the type served by this factory.

Throws OffsetOutOfBoundsException if the base is negative or greater than the size of the raw memory area.

```
@SCJAllowed(javax.safetycritical.annotate.Level.LEVEL_0)
@SCJRestricted({javax.safetycritical.annotate.Phase.
   INTERRUPT_SERVICE_ROUTINE}, mayAllocate = false, maySelfSuspend = false)
public javax.realtime.RawLongArrayRead newRawLongArrayRead(long base,
  int entries)
```

Creates an accessor object for read accessing a long array in raw memory.

returns an object implementing the RawLongArray interface.

Throws AlignmentError if the base is not on the appropriate boundary.

Throws SizeOutOfBoundsException if the long array falls in an invalid address range.

Throws MemoryTypeConflictException if base does not point to memory that matches the type served by this factory.

Throws OffsetOutOfBoundsException if the base is negative or greater than the size of the raw memory area.

```
@SCJAllowed(javax.safetycritical.annotate.Level.LEVEL_0)
@SCJRestricted({javax.safetycritical.annotate.Phase.
   INTERRUPT_SERVICE_ROUTINE}, mayAllocate = false, maySelfSuspend = false)
public javax.realtime.RawLongArrayWrite newRawLongArrayWrite(
  long base,
  int entries)
```

Creates an accessor object for write accessing a long array in raw memory.

returns an object implementing the RawLongArray interface.

Throws AlignmentError if the base is not on the appropriate boundary.

Throws SizeOutOfBoundsException if the long array falls in an invalid address range.

Throws MemoryTypeConflictException if base does not point to memory that matches the type served by this factory.

Throws OffsetOutOfBoundsException if the base is negative or greater than the size of the raw memory area.

```
@SCJAllowed(javax.safetycritical.annotate.Level.LEVEL_0)
@SCJRestricted({javax.safetycritical.annotate.Phase.
   INTERRUPT_SERVICE_ROUTINE}, mayAllocate = false, maySelfSuspend = false)
public javax.realtime.RawLongRead newRawLongRead(long offset)
```

Creates an accessor object for read accessing a long in raw memory.

returns an object implementing the RawLong interface.

Throws AlignmentError if the offset is not on the appropriate boundary.

Throws SizeOutOfBoundsException if the long falls in an invalid address range.

Throws OffsetOutOfBoundsException if the offset is negative or greater than the size of the raw memory area.

Throws MemoryTypeConflictException if offset does not point to memory that matches the type served by this factory.

```
@SCJAllowed(javax.safetycritical.annotate.Level.LEVEL_0)
@SCJRestricted({javax.safetycritical.annotate.Phase.
   INTERRUPT_SERVICE_ROUTINE}, mayAllocate = false, maySelfSuspend = false)
public javax.realtime.RawLongWrite newRawLongWrite(long offset)
```

Creates an accessor object for write accessing a long in raw memory.

returns an object implementing the RawLong interface.

Throws AlignmentError if the offset is not on the appropriate boundary.

Throws SizeOutOfBoundsException if the long falls in an invalid address range.

Throws OffsetOutOfBoundsException if the offset is negative or greater than the size of the raw memory area.

Throws MemoryTypeConflictException if offset does not point to memory that matches the type served by this factory.

```
@SCJAllowed(javax.safetycritical.annotate.Level.LEVEL_0)
@SCJRestricted({javax.safetycritical.annotate.Phase.
   INTERRUPT_SERVICE_ROUTINE}, mayAllocate = false, maySelfSuspend = false)
public javax.realtime.RawShort newRawShort(long offset)
```

Creates an accessor object for accessing a short in raw memory.

Throws AlignmentError if the offset is not on the appropriate boundary.

returns an object implementing the RawShort interface.

Throws SizeOutOfBoundsException if the short falls in an invalid address range.

Throws OffsetOutOfBoundsException if the offset is negative or greater than the size of the raw memory area.

Throws MemoryTypeConflictException if offset does not point to memory that matches the type served by this factory.

```
@SCJAllowed(javax.safetycritical.annotate.Level.LEVEL_0)
@SCJRestricted({javax.safetycritical.annotate.Phase.
   INTERRUPT_SERVICE_ROUTINE}, mayAllocate = false, maySelfSuspend = false)
public javax.realtime.RawShortArray newRawShortArray(long base,
  int entries)
```

Creates an accessor object for accessing a short array in raw memory.

returns an object implementing the RawShortArray interface.

Throws AlignmentError if the base is not on the appropriate boundary.

Throws SizeOutOfBoundsException if the short array falls in an invalid address range.

Throws MemoryTypeConflictException if base does not point to memory that matches the type served by this factory.

Throws OffsetOutOfBoundsException if the base is negative or greater than the size of the raw memory area.

```
@SCJAllowed(javax.safetycritical.annotate.Level.LEVEL_0)
@SCJRestricted({javax.safetycritical.annotate.Phase.
    INTERRUPT_SERVICE_ROUTINE}, mayAllocate = false, maySelfSuspend = false)
public javax.realtime.RawShortArrayRead newRawShortArrayRead(
  long base,
  int entries)
```

Creates an accessor object for read accessing a short array in raw memory.

returns an object implementing the RawShortArray interface.

Throws AlignmentError if the base is not on the appropriate boundary.

Throws SizeOutOfBoundsException if the short array falls in an invalid address range.

Throws MemoryTypeConflictException if base does not point to memory that matches the type served by this factory.

Throws OffsetOutOfBoundsException if the base is negative or greater than the size of the raw memory area.

```
@SCJAllowed(javax.safetycritical.annotate.Level.LEVEL_0)
@SCJRestricted({javax.safetycritical.annotate.Phase.
    INTERRUPT_SERVICE_ROUTINE}, mayAllocate = false, maySelfSuspend = false)
public javax.realtime.RawShortArrayWrite newRawShortArrayWrite(
  long base,
  int entries)
```

Creates an accessor object for write accessing a short array in raw memory.

returns an object implementing the RawShortArray interface.

Throws AlignmentError if the base is not on the appropriate boundary.

Throws SizeOutOfBoundsException if the short array falls in an invalid address range.

Throws MemoryTypeConflictException if base does not point to memory that matches the type served by this factory.

Throws OffsetOutOfBoundsException if the base is negative or greater than the size of the raw memory area.

@SCJAllowed(javax.safetycritical.annotate.Level.LEVEL_0)
@SCJRestricted({javax.safetycritical.annotate.Phase.
INTERRUPT_SERVICE_ROUTINE}, mayAllocate = false, maySelfSuspend = false)
public javax.realtime.RawShortRead newRawShortRead(**long** offset)

Creates an accessor object for read accessing a short in raw memory.

Throws AlignmentError if the offset is not on the appropriate boundary.

returns an object implementing the RawShort interface.

Throws SizeOutOfBoundsException if the short falls in an invalid address range.

Throws OffsetOutOfBoundsException if the offset is negative or greater than the size of the raw memory area.

Throws MemoryTypeConflictException if offset does not point to memory that matches the type served by this factory.

@SCJAllowed(javax.safetycritical.annotate.Level.LEVEL_0)
@SCJRestricted({javax.safetycritical.annotate.Phase.
INTERRUPT_SERVICE_ROUTINE}, mayAllocate = false, maySelfSuspend = false)
public javax.realtime.RawShortWrite newRawShortWrite(**long** offset)

Creates an accessor object for write accessing a short in raw memory.

returns an object implementing the RawShort interface.

Throws AlignmentError if the offset is not on the appropriate boundary.

Throws SizeOutOfBoundsException if the short falls in an invalid address range.

Throws OffsetOutOfBoundsException if the offset is negative or greater than the size of the raw memory area.

Throws MemoryTypeConflictException if offset does not point to memory that matches the type served by this factory.

5.1.29 javax.realtime.RawMemory

Declaration

@SCJAllowed(javax.safetycritical.annotate.Level.LEVEL_0)
public final class RawMemory **extends** java.lang.Object

Description

This class is the hub of a system that constructs special-purpose objects that access particular types and ranges of raw memory. This facility is supported by the registerRawIntegralAccessFactory and the create methods. Four raw-integral-access factories are supported: two for accessing memory (called IO_PORT_MAPPED and IO_MEMORY_MAPPED), one for accessing memory that can be used for DMA (called DMA_ACCESS) and the other for accesses to the memory (called MEM_ACCESS).

Fields

@SCJAllowed(javax.safetycritical.annotate.Level.LEVEL_0)
public static final javax.realtime.RawMemoryName DMA_ACCESS

The name indicating an area of raw memory which is accessable for DMA transfer.

@SCJAllowed(javax.safetycritical.annotate.Level.LEVEL_0)
public static final javax.realtime.RawMemoryName IO_MEM_MAPPED

The name indicating an area of raw memory which is used to access memory mapped IO registers.

@SCJAllowed(javax.safetycritical.annotate.Level.LEVEL_0)
public static final javax.realtime.RawMemoryName IO_PORT_MAPPED

The name indicating an area of raw memory which is used as port-mapped IO registers.

@SCJAllowed(javax.safetycritical.annotate.Level.LEVEL_0)
public static final javax.realtime.RawMemoryName MEM_ACCESS

The name indicating an area of raw memory.

Constructors

@SCJAllowed(javax.safetycritical.annotate.Level.LEVEL_0)
public RawMemory( )

Methods

@SCJAllowed(javax.safetycritical.annotate.Level.LEVEL_0)
@SCJRestricted({javax.safetycritical.annotate.Phase.
  INTERRUPT_SERVICE_ROUTINE}, mayAllocate = false, maySelfSuspend = false)
public static javax.realtime.RawByteArray createRawByteArrayInstance(
 RawMemoryName type,
 long base,
 long size)

Creates or finds an accessor object for accessing a byte array in raw memory.

type — is the required type of memory.

base — is the offset of the required array.

size — is the length of the array.

returns an accessor object from the raw memory access.

Throws AlignmentError if the base is not on the appropriate boundary.

Throws SizeOutOfBoundsException if the byte array falls in an invalid address range.

Throws MemoryTypeConflictException if base does not point to memory that matches the type served by this factory.

Throws OffsetOutOfBoundsException if the base is negative or greater than the size of the raw memory area.

```
@SCJAllowed(javax.safetycritical.annotate.Level.LEVEL_0)
@SCJRestricted({javax.safetycritical.annotate.Phase.
    INTERRUPT_SERVICE_ROUTINE}, mayAllocate = false, maySelfSuspend = false)
public static
javax.realtime.RawByteArrayRead createRawByteArrayReadInstance(
  RawMemoryName type,
  long base,
  long size)
```

Creates or finds an accessor object for accessing a byte array in raw memory.

type — is the required type of memory.

base — is the offset of the required array.

size — is the length of the array.

returns an accessor object from the raw memory access.

Throws AlignmentError if the base is not on the appropriate boundary.

Throws SizeOutOfBoundsException if the byte array falls in an invalid address range.

Throws MemoryTypeConflictException if base does not point to memory that matches the type served by this factory.

Throws OffsetOutOfBoundsException if the base is negative or greater than the size of the raw memory area.

```
@SCJAllowed(javax.safetycritical.annotate.Level.LEVEL_0)
@SCJRestricted({javax.safetycritical.annotate.Phase.
   INTERRUPT_SERVICE_ROUTINE}, mayAllocate = false, maySelfSuspend = false)
public static
javax.realtime.RawByteArrayWrite createRawByteArrayWriteInstance(
  RawMemoryName type,
  long base,
  long size)
```

Creates or finds an accessor object for accessing a byte array in raw memory.

type — is the required type of memory.

base — is the offset of the required array.

size — is the length of the array.

returns an accessor object from the raw memory access.

Throws AlignmentError if the base is not on the appropriate boundary.

Throws SizeOutOfBoundsException if the byte array falls in an invalid address range.

Throws MemoryTypeConflictException if base does not point to memory that matches the type served by this factory.

Throws OffsetOutOfBoundsException if the base is negative or greater than the size of the raw memory area.

```
@SCJAllowed(javax.safetycritical.annotate.Level.LEVEL_0)
@SCJRestricted({javax.safetycritical.annotate.Phase.
   INTERRUPT_SERVICE_ROUTINE}, mayAllocate = false, maySelfSuspend = false)
public static javax.realtime.RawByte createRawByteInstance(
  RawMemoryName type,
  long base)
```

Creates or finds an accessor object for accessing a byte of raw memory.

type — is the required type of memory.

base — is the offset of the required byte.

returns an accessor object from the raw memory access.

Throws AlignmentError if the base is not on the appropriate boundary.

Throws SizeOutOfBoundsException if the long array falls in an invalid address range.

Throws MemoryTypeConflictException if base does not point to memory that matches the type served by this factory.

Throws OffsetOutOfBoundsException if the base is negative or greater than the size of the raw memory area.

```
@SCJAllowed(javax.safetycritical.annotate.Level.LEVEL_0)
@SCJRestricted({javax.safetycritical.annotate.Phase.
   INTERRUPT_SERVICE_ROUTINE}, mayAllocate = false, maySelfSuspend = false)
public static javax.realtime.RawByteRead createRawByteReadInstance(
  RawMemoryName type,
  long base)
```

Creates or finds an accessor object for accessing a byte of raw memory.

type — is the required type of memory.

base — is the offset of the required byte.

returns an accessor object from the raw memory access.

Throws AlignmentError if the base is not on the appropriate boundary.

Throws SizeOutOfBoundsException if the long array falls in an invalid address range.

Throws MemoryTypeConflictException if base does not point to memory that matches the type served by this factory.

Throws OffsetOutOfBoundsException if the base is negative or greater than the size of the raw memory area.

```
@SCJAllowed(javax.safetycritical.annotate.Level.LEVEL_0)
@SCJRestricted({javax.safetycritical.annotate.Phase.
   INTERRUPT_SERVICE_ROUTINE}, mayAllocate = false, maySelfSuspend = false)
public static javax.realtime.RawByteWrite createRawByteWriteInstance(
  RawMemoryName type,
  long base)
```

Creates or finds an accessor object for accessing a byte of raw memory.

type — is the required type of memory.

base — is the offset of the required byte.

returns an accessor object from the raw memory access.

Throws AlignmentError if the base is not on the appropriate boundary.

Throws SizeOutOfBoundsException if the long array falls in an invalid address range.

Throws MemoryTypeConflictException if base does not point to memory that matches the type served by this factory.

Throws OffsetOutOfBoundsException if the base is negative or greater than the size of the raw memory area.

```
@SCJAllowed(javax.safetycritical.annotate.Level.LEVEL_0)
@SCJRestricted({javax.safetycritical.annotate.Phase.
   INTERRUPT_SERVICE_ROUTINE}, mayAllocate = false, maySelfSuspend = false)
public static javax.realtime.RawIntArray createRawIntArrayInstance(
  RawMemoryName type,
  long base,
  long size)
```

Creates or finds an accessor object for accessing a int array in raw memory.

type — is the required type of memory.

base — is the offset of the required int array.

size — is the length of the array.

returns an accessor object from the raw memory access.

Throws AlignmentError if the base is not on the appropriate boundary.

Throws SizeOutOfBoundsException if the short array falls in an invalid address range.

Throws MemoryTypeConflictException if base does not point to memory that matches the type served by this factory.

Throws OffsetOutOfBoundsException if the base is negative or greater than the size of the raw memory area.

```
@SCJAllowed(javax.safetycritical.annotate.Level.LEVEL_0)
@SCJRestricted({javax.safetycritical.annotate.Phase.
   INTERRUPT_SERVICE_ROUTINE}, mayAllocate = false, maySelfSuspend = false)
public static
javax.realtime.RawIntArrayRead createRawIntArrayReadInstance(
  RawMemoryName type,
  long base,
  long size)
```

Creates or finds an accessor object for read accessing a int array in raw memory.

type — is the required type of memory.

base — is the offset of the required int array.

size — is the length of the array.

returns an accessor object from the raw memory access.

Throws AlignmentError if the base is not on the appropriate boundary.

Throws SizeOutOfBoundsException if the short array falls in an invalid address range.

Throws MemoryTypeConflictException if base does not point to memory that matches the type served by this factory.

Throws OffsetOutOfBoundsException if the base is negative or greater than the size of the raw memory area.

@SCJAllowed(javax.safetycritical.annotate.Level.LEVEL_0)
@SCJRestricted({javax.safetycritical.annotate.Phase.
    INTERRUPT_SERVICE_ROUTINE}, mayAllocate = false, maySelfSuspend = false)
public static
javax.realtime.RawIntArrayWrite createRawIntArrayWriteInstance(
  RawMemoryName type,
  long base,
  long size)

Creates or finds an accessor object for write accessing a int array in raw memory.

type — is the required type of memory.

base — is the offset of the required int array.

size — is the length of the array.

returns an accessor object from the raw memory access.

Throws AlignmentError if the base is not on the appropriate boundary.

Throws SizeOutOfBoundsException if the short array falls in an invalid address range.

Throws MemoryTypeConflictException if base does not point to memory that matches the type served by this factory.

Throws OffsetOutOfBoundsException if the base is negative or greater than the size of the raw memory area.

@SCJAllowed(javax.safetycritical.annotate.Level.LEVEL_0)
@SCJRestricted({javax.safetycritical.annotate.Phase.
    INTERRUPT_SERVICE_ROUTINE}, mayAllocate = false, maySelfSuspend = false)
public static javax.realtime.RawInt createRawIntInstance(
  RawMemoryName type,
  long base)

Creates or finds an accessor object for accessing a int of raw memory.

type — is the required type of memory.

base — is the offset of the required int.

returns an accessor object from the raw memory access.

Throws AlignmentError if the base is not on the appropriate boundary.

Throws SizeOutOfBoundsException if the int falls in an invalid address range.

Throws MemoryTypeConflictException if base does not point to memory that matches the type served by this factory.

Throws OffsetOutOfBoundsException if the base is negative or greater than the size of the raw memory area.

```
@SCJAllowed(javax.safetycritical.annotate.Level.LEVEL_0)
@SCJRestricted({javax.safetycritical.annotate.Phase.
   INTERRUPT_SERVICE_ROUTINE}, mayAllocate = false, maySelfSuspend = false)
public static javax.realtime.RawIntRead createRawIntReadInstance(
  RawMemoryName type,
  long base)
```

Creates or finds an accessor object for read accessing a int of raw memory.

type — is the required type of memory.

base — is the offset of the required int.

returns an accessor object from the raw memory access.

Throws AlignmentError if the base is not on the appropriate boundary.

Throws SizeOutOfBoundsException if the int falls in an invalid address range.

Throws MemoryTypeConflictException if base does not point to memory that matches the type served by this factory.

Throws OffsetOutOfBoundsException if the base is negative or greater than the size of the raw memory area.

```
@SCJAllowed(javax.safetycritical.annotate.Level.LEVEL_0)
@SCJRestricted({javax.safetycritical.annotate.Phase.
   INTERRUPT_SERVICE_ROUTINE}, mayAllocate = false, maySelfSuspend = false)
public static javax.realtime.RawIntWrite createRawIntWriteInstance(
  RawMemoryName type,
  long base)
```

Creates or finds an accessor object for write accessing a int of raw memory.

type — is the required type of memory.

base — is the offset of the required int.

returns an accessor object from the raw memory access.

Throws AlignmentError if the base is not on the appropriate boundary.

Throws SizeOutOfBoundsException if the int falls in an invalid address range.

Throws MemoryTypeConflictException if base does not point to memory that matches the type served by this factory.

Throws OffsetOutOfBoundsException if the base is negative or greater than the size of the raw memory area.

```
@SCJAllowed(javax.safetycritical.annotate.Level.LEVEL_0)
@SCJRestricted({javax.safetycritical.annotate.Phase.
   INTERRUPT_SERVICE_ROUTINE}, mayAllocate = false, maySelfSuspend = false)
public static javax.realtime.RawLongArray createRawLongArrayInstance(
  RawMemoryName type,
  long base,
  long size)
```

Creates or finds an accessor object for accessing a long array in raw memory.

type — is the required type of memory.

base — is the offset of the required long array.

size — is the length of the array.

returns an accessor object from the raw memory access.

Throws AlignmentError if the base is not on the appropriate boundary.

Throws SizeOutOfBoundsException if the long array falls in an invalid address range.

Throws MemoryTypeConflictException if base does not point to memory that matches the type served by this factory.

Throws OffsetOutOfBoundsException if the base is negative or greater than the size of the raw memory area.

```
@SCJAllowed(javax.safetycritical.annotate.Level.LEVEL_0)
@SCJRestricted({javax.safetycritical.annotate.Phase.
   INTERRUPT_SERVICE_ROUTINE}, mayAllocate = false, maySelfSuspend = false)
public static
javax.realtime.RawLongArrayRead createRawLongArrayReadInstance(
  RawMemoryName type,
  long base,
  long size)
```

Creates or finds an accessor object for read accessing a long array in raw memory.

type — is the required type of memory.

base — is the offset of the required long array.

size — is the length of the array.

returns an accessor object from the raw memory access.

Throws AlignmentError if the base is not on the appropriate boundary.

Throws SizeOutOfBoundsException if the long array falls in an invalid address range.

Throws MemoryTypeConflictException if base does not point to memory that matches the type served by this factory.

Throws OffsetOutOfBoundsException if the base is negative or greater than the size of the raw memory area.

```
@SCJAllowed(javax.safetycritical.annotate.Level.LEVEL_0)
@SCJRestricted({javax.safetycritical.annotate.Phase.
   INTERRUPT_SERVICE_ROUTINE}, mayAllocate = false, maySelfSuspend = false)
public static
javax.realtime.RawLongArrayWrite createRawLongArrayWriteInstance(
  RawMemoryName type,
  long base,
  long size)
```

Creates or finds an accessor object for write accessing a long array in raw memory.

type — is the required type of memory.

base — is the offset of the required long array.

size — is the length of the array.

returns an accessor object from the raw memory access.

Throws AlignmentError if the base is not on the appropriate boundary.

Throws SizeOutOfBoundsException if the long array falls in an invalid address range.

Throws MemoryTypeConflictException if base does not point to memory that matches the type served by this factory.

Throws OffsetOutOfBoundsException if the base is negative or greater than the size of the raw memory area.

```
@SCJAllowed(javax.safetycritical.annotate.Level.LEVEL_0)
@SCJRestricted({javax.safetycritical.annotate.Phase.
   INTERRUPT_SERVICE_ROUTINE}, mayAllocate = false, maySelfSuspend = false)
```

```
public static javax.realtime.RawLong createRawLongInstance(
  RawMemoryName type,
  long base)
```

Creates or finds an accessor object for accessing a long in raw memory.

type — is the required type of memory.

base — is the offset of the required long.

returns an accessor object from the raw memory access.

Throws AlignmentError if the base is not on the appropriate boundary.

Throws SizeOutOfBoundsException if the long falls in an invalid address range.

Throws MemoryTypeConflictException if base does not point to memory that matches the type served by this factory.

Throws OffsetOutOfBoundsException if the base is negative or greater than the size of the raw memory area.

```
@SCJAllowed(javax.safetycritical.annotate.Level.LEVEL_0)
@SCJRestricted({javax.safetycritical.annotate.Phase.
   INTERRUPT_SERVICE_ROUTINE}, mayAllocate = false, maySelfSuspend = false)
public static javax.realtime.RawLongRead createRawLongReadInstance(
  RawMemoryName type,
  long base)
```

Creates or finds an accessor object for read accessing a long in raw memory.

type — is the required type of memory.

base — is the offset of the required long.

returns an accessor object from the raw memory access.

Throws AlignmentError if the base is not on the appropriate boundary.

Throws SizeOutOfBoundsException if the long falls in an invalid address range.

Throws MemoryTypeConflictException if base does not point to memory that matches the type served by this factory.

Throws OffsetOutOfBoundsException if the base is negative or greater than the size of the raw memory area.

```
@SCJAllowed(javax.safetycritical.annotate.Level.LEVEL_0)
@SCJRestricted({javax.safetycritical.annotate.Phase.
   INTERRUPT_SERVICE_ROUTINE}, mayAllocate = false, maySelfSuspend = false)
public static javax.realtime.RawLongWrite createRawLongWriteInstance(
  RawMemoryName type,
  long base)
```

Creates or finds an accessor object for write accessing a long in raw memory.

type — is the required type of memory.

base — is the offset of the required long.

returns an accessor object from the raw memory access.

Throws AlignmentError if the base is not on the appropriate boundary.

Throws SizeOutOfBoundsException if the long falls in an invalid address range.

Throws MemoryTypeConflictException if base does not point to memory that matches the type served by this factory.

Throws OffsetOutOfBoundsException if the base is negative or greater than the size of the raw memory area.

```
@SCJAllowed(javax.safetycritical.annotate.Level.LEVEL_0)
@SCJRestricted({javax.safetycritical.annotate.Phase.
   INTERRUPT_SERVICE_ROUTINE}, mayAllocate = false, maySelfSuspend = false)
public static javax.realtime.RawShortArray createRawShortArrayInstance(
  RawMemoryName type,
  long base,
  long size)
```

Creates or finds an accessor object for accessing a short array in raw memory.

type — is the required type of memory.

base — is the offset of the required short array.

size — is the length of the array.

returns an accessor object from the raw memory access.

Throws AlignmentError if the base is not on the appropriate boundary.

Throws SizeOutOfBoundsException if the short array falls in an invalid address range.

Throws MemoryTypeConflictException if base does not point to memory that matches the type served by this factory.

Throws OffsetOutOfBoundsException if the base is negative or greater than the size of the raw memory area.

```
@SCJAllowed(javax.safetycritical.annotate.Level.LEVEL_0)
@SCJRestricted({javax.safetycritical.annotate.Phase.
   INTERRUPT_SERVICE_ROUTINE}, mayAllocate = false, maySelfSuspend = false)
public static
javax.realtime.RawShortArrayRead createRawShortArrayReadInstance(
  RawMemoryName type,
  long base,
  long size)
```

Creates or finds an accessor object for read accessing a short array in raw memory.

type — is the required type of memory.

base — is the offset of the required short array.

size — is the length of the array.

returns an accessor object from the raw memory access.

Throws AlignmentError if the base is not on the appropriate boundary.

Throws SizeOutOfBoundsException if the short array falls in an invalid address range.

Throws MemoryTypeConflictException if base does not point to memory that matches the type served by this factory.

Throws OffsetOutOfBoundsException if the base is negative or greater than the size of the raw memory area.

```
@SCJAllowed(javax.safetycritical.annotate.Level.LEVEL_0)
@SCJRestricted({javax.safetycritical.annotate.Phase.
   INTERRUPT_SERVICE_ROUTINE}, mayAllocate = false, maySelfSuspend = false)
public static
javax.realtime.RawShortArrayWrite createRawShortArrayWriteInstance(
  RawMemoryName type,
  long base,
  long size)
```

Creates or finds an accessor object for write accessing a short array in raw memory.

type — is the required type of memory.

base — is the offset of the required short array.

size — is the length of the array.

returns an accessor object from the raw memory access.

Throws AlignmentError if the base is not on the appropriate boundary.

Throws SizeOutOfBoundsException if the short array falls in an invalid address range.

Throws MemoryTypeConflictException if base does not point to memory that matches the type served by this factory.

Throws OffsetOutOfBoundsException if the base is negative or greater than the size of the raw memory area.

```
@SCJAllowed(javax.safetycritical.annotate.Level.LEVEL_0)
@SCJRestricted({javax.safetycritical.annotate.Phase.
   INTERRUPT_SERVICE_ROUTINE}, mayAllocate = false, maySelfSuspend = false)
public static javax.realtime.RawShort createRawShortInstance(
  RawMemoryName type,
  long base)
```

Creates or finds an accessor object for accessing a short of raw memory.

type — is the required type of memory.

base — is the offset of the required short.

returns an accessor object from the raw memory access.

Throws AlignmentError if the base is not on the appropriate boundary.

Throws SizeOutOfBoundsException if the short falls in an invalid address range.

Throws MemoryTypeConflictException if base does not point to memory that matches the type served by this factory.

Throws OffsetOutOfBoundsException if the base is negative or greater than the size of the raw memory area.

```
@SCJAllowed(javax.safetycritical.annotate.Level.LEVEL_0)
@SCJRestricted({javax.safetycritical.annotate.Phase.
   INTERRUPT_SERVICE_ROUTINE}, mayAllocate = false, maySelfSuspend = false)
public static javax.realtime.RawShortRead createRawShortReadInstance(
  RawMemoryName type,
  long base)
```

Creates or finds an accessor object for read accessing a short of raw memory.

type — is the required type of memory.

base — is the offset of the required short.

returns an accessor object from the raw memory access.

Throws AlignmentError if the base is not on the appropriate boundary.

Throws SizeOutOfBoundsException if the short falls in an invalid address range.

Throws MemoryTypeConflictException if base does not point to memory that matches the type served by this factory.

Throws OffsetOutOfBoundsException if the base is negative or greater than the size of the raw memory area.

```
@SCJAllowed(javax.safetycritical.annotate.Level.LEVEL_0)
@SCJRestricted({javax.safetycritical.annotate.Phase.
   INTERRUPT_SERVICE_ROUTINE}, mayAllocate = false, maySelfSuspend = false)
public static javax.realtime.RawShortWrite createRawShortWriteInstance(
  RawMemoryName type,
  long base)
```

Creates or finds an accessor object for write accessing a short of raw memory.

type — is the required type of memory.

base — is the offset of the required short.

returns an accessor object from the raw memory access.

Throws AlignmentError if the base is not on the appropriate boundary.

Throws SizeOutOfBoundsException if the short falls in an invalid address range.

Throws MemoryTypeConflictException if base does not point to memory that matches the type served by this factory.

Throws OffsetOutOfBoundsException if the base is negative or greater than the size of the raw memory area.

```
@SCJAllowed(javax.safetycritical.annotate.Level.LEVEL_0)
@SCJRestricted(mayAllocate = false, maySelfSuspend = false)
public static void registerAccessFactory(
  RawIntegralAccessFactory factory)
```

Registers a factory for accessing raw memory.

factory — is the factory being registered.

Throws java.lang.IllegalArgumentException if factory is null or its name is served by a factory that has already been registered.

5.2 Interrupt Handling

Unlike 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 program.
- Delivery is the action that invokes a *registered* interrupt service routine (ISR) in response to the occurrence 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 occurrences may be inhibited. While an interrupt occurrence is inhibited, all occurrences of that interrupt shall be prevented from being delivered. 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 time keeping by the JVM, 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. Any daisy-chaining of interrupt handlers shall be performed explicitly by the application interrupt handlers.
- 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 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, what is the mechanism 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 should 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 RawIntegralAccess interface,
- AsynchronousEventHandler release.

SCJ requires that all methods that can be called from an ISR object shall be annotated with @SCJRestricted(INTERRUPT_SERVICE_ROUTINE). An implementations shall provide a list of all such implementation-provided interrupt-safe methods.

SCJ defines the notion of interrupt priorities (see Section 4.6.5). 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.

5.2.1 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.

5.2.2 javax.realtime.InterruptServiceRoutine

This class is a restricted version of the class provided by the RTSJ version 1.1 specification.

Declaration

@SCJAllowed(javax.safetycritical.annotate.Level.LEVEL_1)
public abstract class InterruptServiceRoutine **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.

Methods

@SCJAllowed(javax.safetycritical.annotate.Level.LEVEL_1)
public static javax.realtime.InterruptServiceRoutine getISR(
  int interrupt)

returns the ISR registered with the given interrupt. Null is returned if nothing is registered.

@SCJAllowed(javax.safetycritical.annotate.Level.LEVEL_1)
public static javax.realtime.AffinitySet getInterruptAffinity(
  int InterruptId)

Every interrupt has an affinity that indicates which processors might service a hardware interrupt request. The returned set is preallocated and resides in immortal memory.

returns The affinity set of the processors.

Throws IllegalArgument if unsupported InterruptId

@SCJAllowed(javax.safetycritical.annotate.Level.LEVEL_1)
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 IllegalArgument if unsupported InterruptId

@SCJAllowed(javax.safetycritical.annotate.Level.LEVEL_1)
public final java.lang.String getName( )

Get the name of this interrupt service routine.

returns the name of this interrupt service routine.

@SCJAllowed(javax.safetycritical.annotate.Level.LEVEL_1)
@SCJRestricted({javax.safetycritical.annotate.Phase.
INTERRUPT_SERVICE_ROUTINE}, maySelfSuspend = false)
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. Unless the overridden method is synchronized, the infrastructure shall provide no synchronization for the execution of this method.

5.2.3 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.InterruptServiceRoutine

Constructors

@SCJAllowed(javax.safetycritical.annotate.Level.LEVEL_1)
public ManagedInterruptServiceRoutine(**long** sizes)

Creates an interrupt service routine with the given name and associated with a given interrupt.

sizes — defines the memory space required by the handle method.

initialMemoryAreaSize — is the size of a private memory area which acts as the initial allocation context for the handle method. A size of 0 indicates that any use of the new operator within the initial allocation context will result in an OutOfMemoryException being thrown.

Methods

@SCJAllowed(javax.safetycritical.annotate.Level.LEVEL_1)
@Override
@SCJRestricted({javax.safetycritical.annotate.Phase.INITIALIZATION})
public final void register(**int** interrupt)
throws javax.realtime.RegistrationException

Equivalent to register(interrupt, highestInterruptCeilingPriority).

@SCJAllowed(javax.safetycritical.annotate.Level.LEVEL_1)
@SCJRestricted({javax.safetycritical.annotate.Phase.INITIALIZATION})
public final void register(**int** interrupt, **int** ceiling)
throws javax.realtime.RegistrationException

Registers the ISR for the given interrupt with the current mission, 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 required ceiling is not as high or higher than this interrupt priority.

@SCJAllowed(javax.safetycritical.annotate.Level.LEVEL_1)
@SCJRestricted({javax.safetycritical.annotate.Phase.
INTERRUPT_SERVICE_ROUTINE})
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)
@SCJRestricted({javax.safetycritical.annotate.Phase.CLEANUP})
public final void unregister( )

Unregisters the ISR with the current mission.

5.2.4 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 address 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 virtual machine 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.

5.3 POSIX Signal Handlers

In the RTSJ, all asynchronous external events are associated with *happenings*. In SCJ, only operating system signals are supported.

5.3.1 Semantics and Requirements

An implementation shall predefine all the POSIX signals and Real-time signals that it supports. Each signal has an associated integer id and a string name. The string name is the name given to the corresponding signal in the POSIX IEEE Std 1003.1-2008.

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.realtime.Happening

Declaration

@SCJAllowed(javax.safetycritical.annotate.Level.LEVEL_1)
public interface Happening

Description

This interface is the common parent type for happenings that are to be used for triggering external events. Happening represent events that can be triggered based on some event external to the VM. This includes POSIX signals and POSIX realtime signals.

Methods

@SCJAllowed(javax.safetycritical.annotate.Level.LEVEL_1)
public void attach(AbstractAsyncEvent event)
  throws java.lang.IllegalArgumentException

Bind an **javax.realtime.AbstractAsyncEvent** to this Happening.

An **javax.realtime.AbstractAsyncEvent** may be bound to more than one signal, and it may be both bound to happenings using AbstractAsyncEvent.bindTo(String) and this method. Also, each signal may be attached to more than one instances of **javax.realtime.AbstractAsyncEvent**

event — to be attached to this signal.

Throws DuplicateEventException when the given event is already attached to the signal

```
@SCJAllowed(javax.safetycritical.annotate.Level.LEVEL_1)
@SCJRestricted(mayAllocate = false,
   maySelfSuspend = false,
   {javax.safetycritical.annotate.Phase.ALL})
```
public void deregister( )

```
@SCJAllowed(javax.safetycritical.annotate.Level.LEVEL_1)
```
public boolean deregisterName( )

Unregister this signal so that it cannot be triggered.

```
@SCJAllowed(javax.safetycritical.annotate.Level.LEVEL_1)
@SCJRestricted(mayAllocate = false,
   maySelfSuspend = false,
   {javax.safetycritical.annotate.Phase.ALL})
```
public int getId( )

Get the ID of this signal.

returns the id of this signal

```
@SCJAllowed(javax.safetycritical.annotate.Level.LEVEL_1)
@SCJRestricted(mayAllocate = false,
   maySelfSuspend = false,
   {javax.safetycritical.annotate.Phase.ALL})
```
public java.lang.String getName( )

Get the name of this signal.

returns the name of this signal.

```
@SCJAllowed(javax.safetycritical.annotate.Level.LEVEL_1)
```
public boolean isHappening(String name)

Determine if a signal with a given name is registered.

name — of the signal

returns true when a signal with the given name is registered

```
@SCJAllowed(javax.safetycritical.annotate.Level.LEVEL_1)
public boolean isRegistered( )
```

Indicates if the happening is registered or not.

returns true when the signal is registered

```
@SCJAllowed(javax.safetycritical.annotate.Level.LEVEL_1)
@SCJRestricted(mayAllocate = false,
   maySelfSuspend = false,
   {javax.safetycritical.annotate.Phase.ALL})
public boolean registerName( )
```

Register this signal with the given manager so that it can be triggered.

returns true when the happening is registered

```
@SCJAllowed(javax.safetycritical.annotate.Level.LEVEL_1)
public void unbind(AbstractAsyncEvent event)
  throws javax.realtime.EventNotFoundException
```

Unbind an **javax.realtime.AbstractAsyncEvent** from this signal.

event — to be unbound from this signal.

Throws EventNotFoundException when the given event is not attached to the given signal.

5.3.4 javax.realtime.safetycriticalPOSIXSignalHandler

Declaration

```
@SCJAllowed(javax.safetycritical.annotate.Level.LEVEL_1)
public abstract class POSIXSignalHandler

  extends javax.safetycritical.ManagedEventHandler
```

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.

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)
@SCJRestricted({javax.safetycritical.annotate.Phase.INITIALIZATION})
public POSIXSignalHandler(PriorityParameters priority,
  AperiodicParameters release,
  StorageParameters storage,
  Happening [] signals)
```

Constructs an real-time POSIX signalt handler that will be released when the signal is delivered.

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

release — specifies the release parameters for this aperiodic event handler; it must not be null.

storage — specifies the storage requirements for this handler

signals — specifies the range of POSIX real-time signals that releases this handler

Throws IllegalArgumentException IllegalArgumentException if priority, release or event is null.

Memory behavior: Does not perform memory allocation. Does not allow this to escape local scope. Builds links from this to priority and parameters, so those two arguments must reside in scopes that enclose this. Builds a link from this to the event, so the event must reside in memory that encloses this.

5.3.5 javax.realtime.safetycriticalPOSIXRealtimeSignalHandler

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.

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)
@SCJRestricted({javax.safetycritical.annotate.Phase.INITIALIZATION})
public POSIXRealtimeSignalHandler(PriorityParameters priority,
  AperiodicParameters release,
  StorageParameters storage,
  Happening [] signals)
```

Constructs an real-time POSIX signalt handler that will be released when the signal is delivered.

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

release — specifies the release parameters for this aperiodic event handler; it must not be null.

storage — specifies the storage requirements for this handler

signals — specifies the range of POSIX real-time signals that releases this handler

Throws IllegalArgumentException IllegalArgumentException if priority, release or event is null.

Memory behavior: Does not perform memory allocation. Does not allow this to escape local scope. Builds links from this to priority and parameters, so those two arguments must reside in scopes that enclose this. Builds a link from this to the event, so the event must reside in memory that encloses this.

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, the former is provided by the notion of *happenings* and the latter via the *physical and raw memory* access facilities. Happenings in the RTSJ do not allow the programmer to write first-level interrupt handlers. SCJ extends the RTSJ model to allow this. 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.

5.5 Compatibility

The SCJ interrupt handling facility uses the same model as the RTSJ.

Chapter 6

Input and Output Model

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. 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 to create a simple, extendable I/O capability.

6.1 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.

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 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.

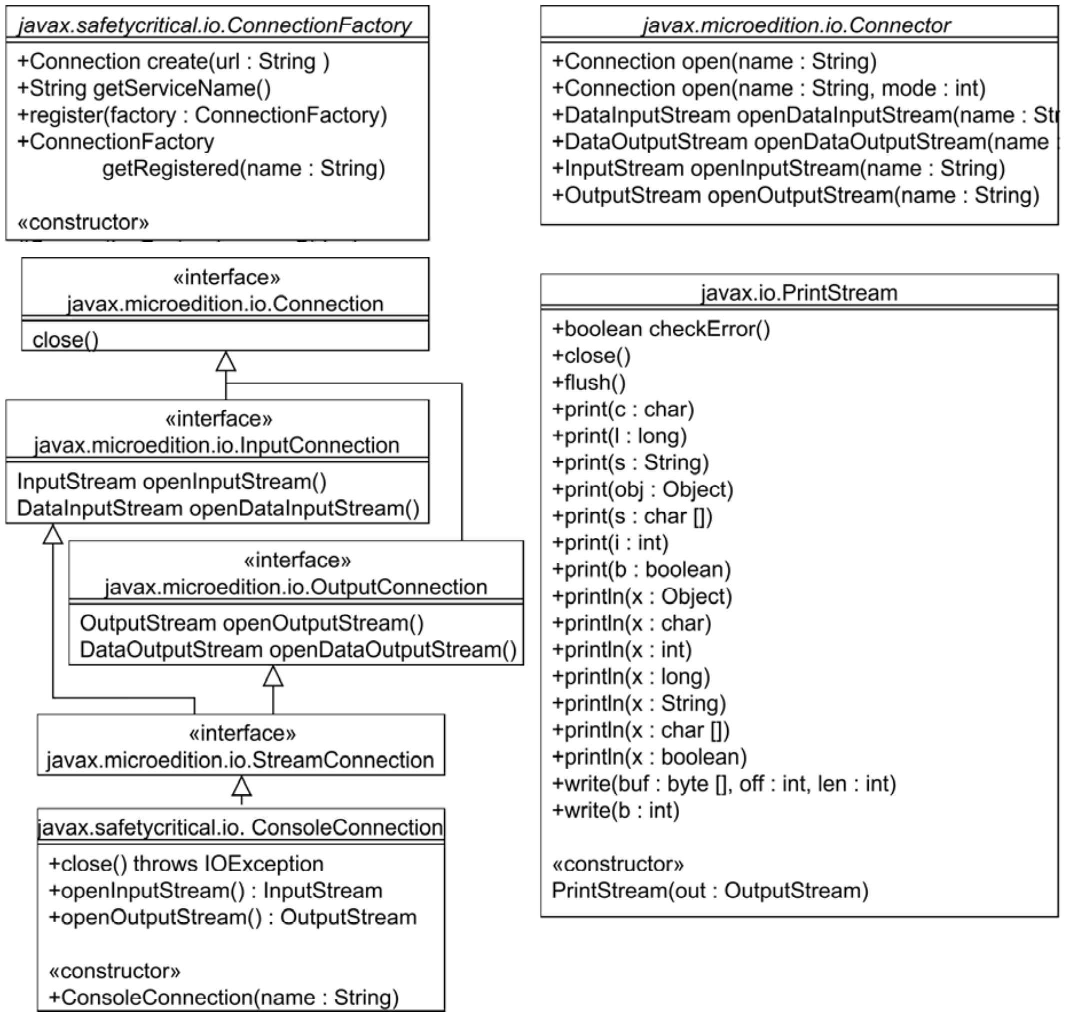
javax.safetycritical.io.ConnectionFactory
+Connection create(url : String)
+String getServiceName()
+register(factory : ConnectionFactory)
+ConnectionFactory getRegistered(name : String)
«constructor»
javax.microedition.io.Connector
+Connection open(name : String)
+Connection open(name : String, mode : int)
+DataInputStream openDataInputStream(name : Str
+DataOutputStream openDataOutputStream(name
+InputStream openInputStream(name : String)
+OutputStream openOutputStream(name : String)
«interface» javax.microedition.io.Connection
close()
«interface» javax.microedition.io.InputConnection
InputStream openInputStream()
DataInputStream openDataInputStream()
«interface» javax.microedition.io.OutputConnection
OutputStream openOutputStream()
DataOutputStream openDataOutputStream()
«interface» javax.microedition.io.StreamConnection
javax.safetycritical.io. ConsoleConnection
+close() throws IOException
+openInputStream() : InputStream
+openOutputStream() : OutputStream
«constructor»
+ConsoleConnection(name : String)
javax.io.PrintStream
+boolean checkError()
+close()
+flush()
+print(c : char)
+print(l : long)
+print(s : String)
+print(obj : Object)
+print(s : char [])
+print(i : int)
+print(b : boolean)
+println(x : Object)
+println(x : char)
+println(x : int)
+println(x : long)
+println(x : String)
+println(x : char [])
+println(x : boolean)
+write(buf : byte [], off : int, len : int)
+write(b : int)
«constructor»
PrintStream(out : OutputStream)

Figure 6.1: Interfaces and classes supporting streaming I/O

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 provided 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.

6.2 Level Considerations

The I/O classes are usable in all SCJ compliance levels.

6.3 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.

6.3.1 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 protocolno timeout 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
public static javax.microedition.io.Connection open(String name)
  throws java.io.IOException

Create and open a Connection.

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 may be thrown if access to the protocol handler is prohibited.

```
@SCJAllowed
public static javax.microedition.io.Connection open(String name,
  int mode)
  throws java.io.IOException
```

Create and open a Connection.

name — The URL for the connection.

mode — The access mode.

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 may be thrown if access to the protocol handler is prohibited.

```
@SCJAllowed
public static java.io.DataInputStream openDataInputStream(String name)
  throws java.io.IOException
```

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 may be thrown if access to the protocol handler is prohibited.

```
@SCJAllowed
public static java.io.DataOutputStream openDataOutputStream(String name)
  throws java.io.IOException
```

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 may be thrown if access to the protocol handler is prohibited.

@SCJAllowed
public static java.io.InputStream openInputStream(String name)
throws java.io.IOException

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 may be thrown if access to the protocol handler is prohibited.

@SCJAllowed
public static java.io.OutputStream openOutputStream(String name)
throws java.io.IOException

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 may be thrown if access to the protocol handler is prohibited.

6.3.2 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
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 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

6.3.3 javax.microedition.io.InputConnection

Declaration

@SCJAllowed
public interface InputConnection
  implements javax.microedition.io.Connection

Description

This interface defines the capabilities that an input stream connection must have.

Methods

@SCJAllowed
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
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.

6.3.4 javax.microedition.io.OutputConnection

Declaration

@SCJAllowed
public interface OutputConnection
implements javax.microedition.io.Connection

Description

This interface defines the capabilities that an output stream connection must have.

Methods

@SCJAllowed
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
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.

6.3.5 javax.microedition.io.StreamConnection

Declaration

@SCJAllowed
public interface StreamConnection
implements 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 (for instance in MIDP), 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.

6.3.6 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
public ConnectionNotFoundException( )

Constructs a ConnectionNotFoundException with no detail message.

@SCJAllowed
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.

6.3.7 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.

Constructors

@SCJAllowed
public ConsoleConnection(String name)
  throws javax.microedition.io.ConnectionNotFoundException

Create a new object of this type.

Methods

@Override @SCJAllowed
public void close( )

Closes this console connection.

@Override @SCJAllowed
public java.io.InputStream openInputStream( )

returns the input stream for this console connection.

@Override @SCJAllowed
public java.io.OutputStream openOutputStream( )

returns the output stream for this console connection.

6.3.8 javax.safetycritical.io.ConnectionFactory

Declaration

@SCJAllowed
public abstract class ConnectionFactory **extends** java.lang.Object

Description

A factory for creating user defined connections.

Constructors

@SCJAllowed
public ConnectionFactory(String name)

Create a connection factory.

name — Connection name used for connection request in Connector.

Methods

@SCJAllowed
public abstract javax.microedition.io.Connection create(String url)
throws java.io.IOException

Create of 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
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
public final java.lang.String getServiceName( )

Return the service name for a connection factory.

returns service name.

@SCJAllowed
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.

6.3.9 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
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
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
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
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
public void print(**int** i)

Print an integer. The string produced by {@link java.lang.String#valueOf(int)} is translated into bytes according to the platform's default character encoding, and these bytes are written in exactly the manner of the **write(int)** method.

i — The int to be printed.

See Also: java.lang.Integer.toString(int)

@SCJAllowed
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(int)** method.

s — The array of chars to be printed.

Throws NullPointerException If s is null

@SCJAllowed
public void print(Object obj)

Print an object. The string produced by the {@link java.lang.String#valueOf(Object)} 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(int)** method.

obj — The Object to be printed.

See Also: java.lang.Object.toString()

@SCJAllowed
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(int)** method.

s — The String to be printed.

@SCJAllowed
public void print(**long** l)

Print a long integer. The string produced by {@link java.lang.String#valueOf(long)} is translated into bytes according to the platform's default character encoding, and these bytes are written in exactly the manner of the **write(int)** method.

l — The long to be printed.

See Also: java.lang.Long.toString(long)

@SCJAllowed
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(int)** method.

c — The char to be printed.

@SCJAllowed
public void print(**boolean** b)

Print a boolean value. The string produced by {@link java.lang.String#valueOf(boolean)} is translated into bytes according to the platform's default character encoding, and these bytes are written in exactly the manner of the **write(int)** method.

b — The boolean to be printed.

@SCJAllowed
public void println(**boolean** x)

Print a boolean and then terminate the line. This method behaves as though it invokes **print(boolean)** and then **println()** .

x — The boolean to be printed.

@SCJAllowed
public void println(**char** x)

Print a character and then terminate the line. This method behaves as though it invokes **print(char)** and then **println()** .

x — The char to be printed.

@SCJAllowed
public void println(**int** x)

Print an integer and then terminate the line. This method behaves as though it invokes **print(int)** and then **println()** .

x — The int to be printed.

@SCJAllowed
public void println(**char** [] x)

Print an array of characters and then terminate the line. This method behaves as though it invokes **print(char[])** and then **println()** .

x — an array of chars to print.

@SCJAllowed
public void println(String x)

Print a String and then terminate the line. This method behaves as though it invokes **print(String)** and then **println()** .

x — The String to be printed.

@SCJAllowed
public void println(Object x)

Print an Object and then terminate the line. This method behaves as though it invokes **print(Object)** and then **println()** .

x — The Object to be printed.

@SCJAllowed
public void println(**long** x)

Print a long and then terminate the line. This method behaves as though it invokes **print(long)** and then **println()** .

x — a The long to be printed.

@SCJAllowed
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
protected void setError( )

Set the error state of the stream to true.

Since
JDK1.1

@SCJAllowed
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(char) or println(char) methods.

buf — A byte array.

off — Offset from which to start taking bytes.

len — Number of bytes to write.

@SCJAllowed
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(char) or println(char) methods.

b — The byte to be written.

See Also: java.io.PrintStream.print(char), java.io.PrintStream.println(char)

6.4 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 within SCJ.

To provide a minimal, standard way to communicate simple text messages, the Java Micro Edition console connection is subsetted within SCJ. This connection provides the ability to report the test results of the TCK on all compliant SCJ implementation.

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.

6.5 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 connection 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

All memory allocation in SCJ, as in standard Java and the RTSJ, is performed from within an *allocation context*. This defines the space where an object is allocated. Whereas standard Java implicitly defines a single heap as the allocation context, the RTSJ generalizes allocation contexts through the classes MemoryArea and the interface AllocationContext. The conventional Java heap is one instance of a subclass of the MemoryArea class.

The RTSJ defines additional allocation contexts including ImmortalMemory and various kinds of ScopedMemory. SCJ restricts the use of these allocation contexts to new subclasses of LTMemory provided by SCJ: MissionMemory and PrivateMemory. In SCJ, instances of these classes are created by the infrastructure; they can not be directly instantiated by the application.

These new allocation contexts all inherit from a common SCJ class, ManagedMemory, which the infrastructure uses to keep track of them. The intent is to reduce the code needed to safely use scoped memory areas. Much of the scope management that needs to be done explicitly in application code in an RTSJ program is done by the SCJ infrastructure.

7.1 Semantics and Requirements

As discussed in Chapter 3, SCJ supports the notion of a mission and a mission life cycle. An SCJ application has three phases as shown in Figure 3.1: initialization, execution, and cleanup.

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 called PrivateMemory during the execution phase. Nested missions are supported, so an application may have more than one active mission memory.

Both MissionMemory and PrivateMemory are direct subclasses of ManagedMemory. As mentioned above, they 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. Unlike the RTSJ, SCJ assigns ownership to these backing store locations. Hence, the backing stores used for a schedulable object's private memory areas are owned by those schedulable objects.

7.1.1 Memory Model

The following defines the requirements for the SCJ memory model.

- 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 linear-time scoped memory area (using the MissionMemory class) 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.
- A private memory area shall be owned by a single schedulable object and it shall be entered only by that schedulable object.
- Every event handler has its own private memory area and all allocations performed during a release (see Chapter 4) of that event handler shall, by default, be performed in this private memory. The memory allocated to objects created in this private memory shall be reclaimed at the end of the release.
- Every thread has its own private memory area and allocations performed during a release (see Chapter 4) of that schedulable object shall, by default, be performed in this 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 by the application and a helper method is provided to ensure this.

- Backing store shall be managed as specified in the StorageParameters 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 is the backing store reservation of its mission sequencer.
- SCJ shall not support object finalizers. A similar effect can be obtained with a try statement that includes a finally clause for any given scope.
- SCJ shall conform to the Java memory model. In addition, all access to raw memory is considered to be volatile access (see Section 5.1).

Figure 7.1 illustrates the use of hierarchical memory areas within SCJ. The diagram shows the scope stacks for six schedulable objects (PEH A .. F). They all share Immortal and Mission memory at their base.

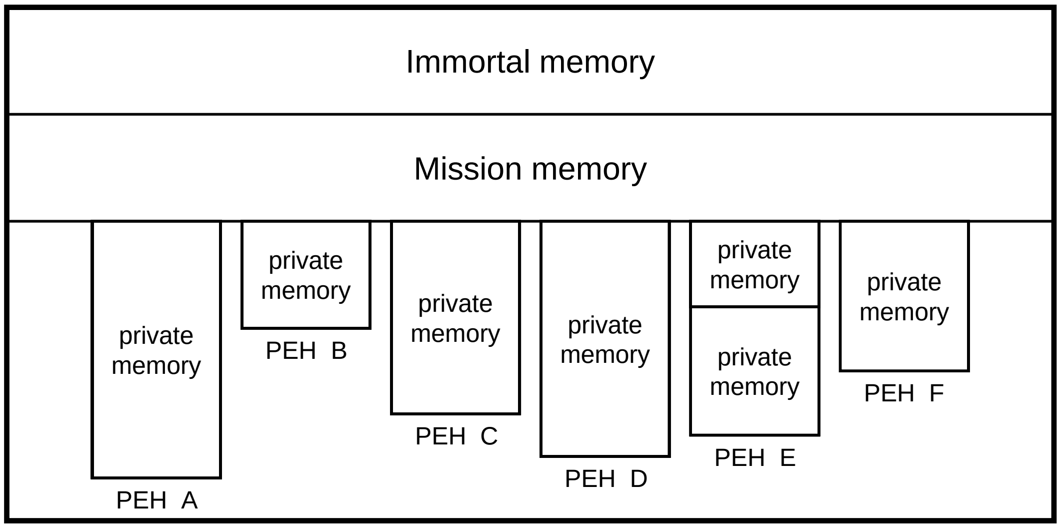

Figure 7.1: Example of Memory Areas used by a Level 1 Application

7.2 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.

7.2.1 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 its own 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.

7.2.2 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.

7.2.3 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 shall not be created when the current allocation context is a private memory. A nested mission memory shall be created by its mission sequencer.

7.3 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.

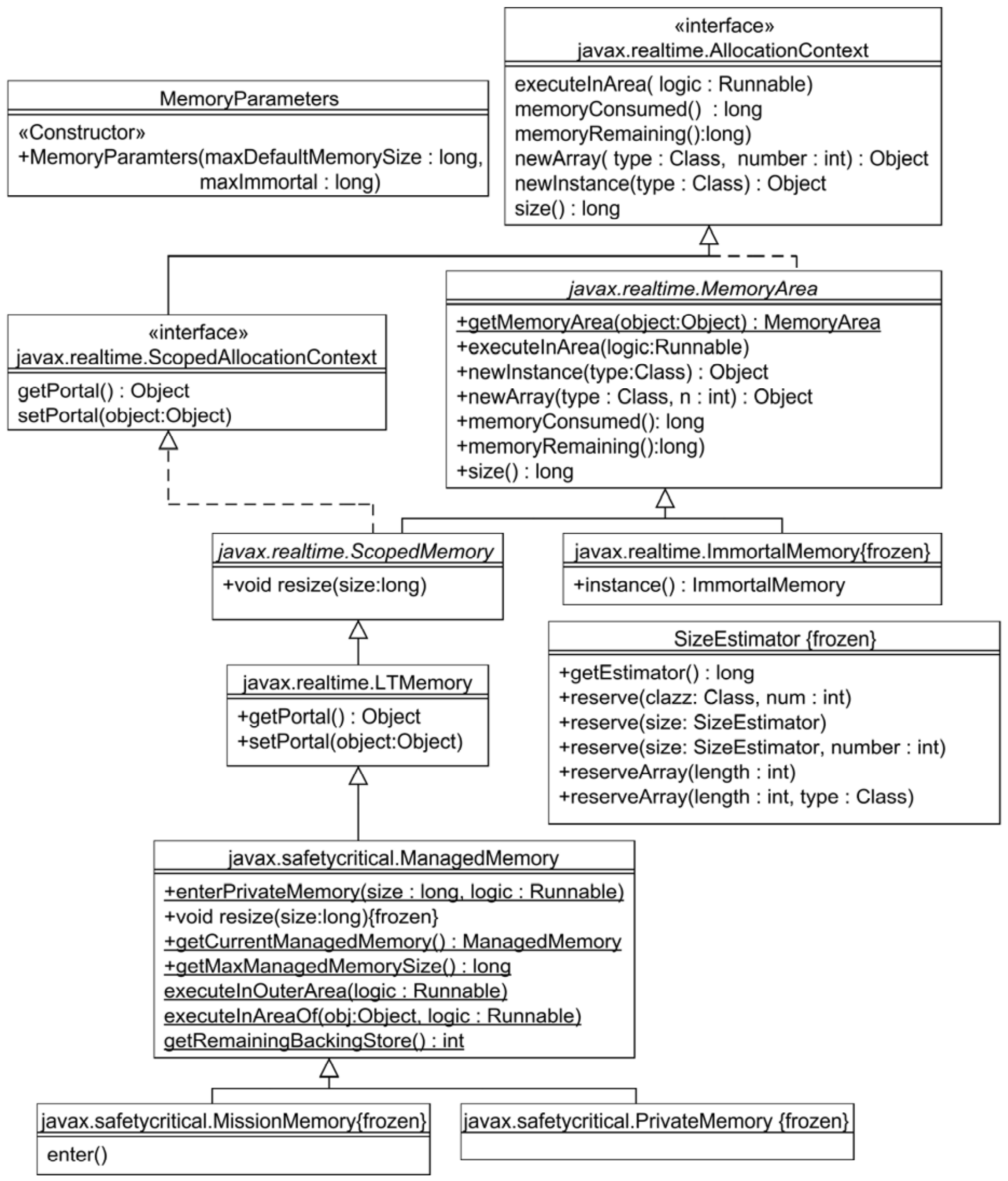
«interface»
javax.realtime.AllocationContext
executeInArea(logic : Runnable)
memoryConsumed() : long
memoryRemaining():long)
newArray(type : Class, number : int) : Object
newInstance(type : Class) : Object
size() : long
MemoryParameters
«Constructor»
+MemoryParamters(maxDefaultMemorySize : long, maxImmortal : long)
javax.realtime.MemoryArea
+getMemoryArea(object:Object) : MemoryArea
+executeInArea(logic:Runnable)
+newInstance(type:Class) : Object
+newArray(type : Class, n : int) : Object
+memoryConsumed(): long
+memoryRemaining():long)
+size() : long
«interface»
javax.realtime.ScopedAllocationContext
getPortal() : Object
setPortal(object:Object)
javax.realtime.ScopedMemory
+void resize(size:long)
javax.realtime.ImmortalMemory{frozen}
+instance() : ImmortalMemory
SizeEstimator {frozen}
+getEstimator() : long
+reserve(clazz: Class, num : int)
+reserve(size: SizeEstimator)
+reserve(size: SizeEstimator, number : int)
+reserveArray(length : int)
+reserveArray(length : int, type : Class)
javax.realtime.LTMemory
+getPortal() : Object
+setPortal(object:Object)
javax.safetycritical.ManagedMemory
+enterPrivateMemory(size : long, logic : Runnable)
+void resize(size:long){frozen}
+getCurrentManagedMemory() : ManagedMemory
+getMaxManagedMemorySize() : long
executeInOuterArea(logic : Runnable)
executeInAreaOf(obj:Object, logic : Runnable)
getRemainingBackingStore() : int
javax.safetycritical.MissionMemory{frozen}
enter()
javax.safetycritical.PrivateMemory {frozen}

Figure 7.2: Overview of MemoryArea-Related Classes

7.3.1 Class javax.realtime.MemoryParameters

Refer to Section 4.3.6

7.3.2 Interface javax.realtime.AllocationContext

Declaration

@SCJAllowed
public interface AllocationContext

Description

All memory allocation takes places from within an allocation context. This interface defines the operations available on all allocation contexts. Allocation contexts are implemented by memory areas. The RTSJenter method is not callable by the SCJ application and hence is omitted.

Methods

@SCJAllowed
public long memoryConsumed( )

returns the amount of memory in bytes consumed so far in this memory area.

@SCJAllowed
public long memoryRemaining( )

returns the amount of memory in bytes remaining in this memory area.

@SCJAllowed @SCJRestricted(maySelfSuspend = false)
public java.lang.Object newArray(**Class**<> type, **int** number)
throws java.lang.IllegalArgumentException

Create an array object of the given type and length number in this memory area.

type — the class of object this memory area should hold. An array of a primitive type can be created using a type such as Integer.TYPE, which would create an array of the int type.

number — the number of elements the array should have.

returns the new array of class type and size number.

Throws IllegalArgumentException if number is less than zero, type is null, or type is java.lang.Void.TYPE.

Throws OutOfMemoryError if space in the memory area is exhausted.

@SCJAllowed
public java.lang.Object newInstance(**Class**<> type)
  throws java.lang.IllegalAccessException,
    java.lang.InstantiationException,
    java.lang.OutOfMemoryError

Create a new instance of class type in this memory area.

type — is the class of the object to be created

returns a new instance of the given class.

Throws IllegalAccessException if the class or constructor is inaccessible due to access rules.

Throws InstantiationException if the specified class object could not be constructed because (1) the class is an interface, (2) the class is abstract, (3) the class is an array, or (4) the class either does not have a no-argument constructor or its no-argument constructor is not visible.

Throws OutOfMemoryError if space in the memory area is exhausted.

@SCJAllowed
public long size( )

returns the current size of this memory area in bytes.

7.3.3 Interface javax.realtime.ScopedAllocationContext

Declaration

@SCJAllowed
public interface ScopedAllocationContext
  implements javax.realtime.AllocationContext

Description

This is the base interface for all scoped memory areas. Scoped memory is a region based memory management strategy that can only be cleared when no thread is executing in the area.

7.3.4 Class javax.realtime.MemoryArea

Declaration

@SCJAllowed
public abstract class MemoryArea
  implements javax.realtime.AllocationContext
  extends java.lang.Object

Description

All allocation contexts are implemented by memory areas. This is the base-level class for all memory areas.

Methods

@SCJAllowed @SCJRestricted(maySelfSuspend = false)
public static javax.realtime.MemoryArea getMemoryArea(Object object)

returns the memory area in which object is allocated,

@SCJAllowed @SCJRestricted(maySelfSuspend = false)
public abstract long memoryConsumed( )

returns the memory consumed in this memory area.

@SCJAllowed @SCJRestricted(maySelfSuspend = false)
public abstract long memoryRemaining( )

returns the memory remaining in this memory area.

@SCJAllowed @SCJRestricted(maySelfSuspend = false)
public java.lang.Object newArray(**Class**<> type, **int** size)
throws javax.realtime.InaccessibleAreaException

This method creates an object of type type in the same memory area that contains this.

type — is the type of the array element for the returned array.

returns a new array of type type with size n objects allocated in this memory area.

@SCJAllowed @SCJRestricted(maySelfSuspend = false)
public java.lang.Object newInstance(**Class**<> type)
throws java.lang.IllegalArgumentException,
java.lang.InstantiationException,
java.lang.OutOfMemoryError,
javax.realtime.InaccessibleAreaException

This method creates an object of type type in this memory area.

type — is the type of the object returned.

returns a new object of type type

Memory behavior: This constructor may allocate objects within the same MemoryArea that holds the argument named "this.area".

@SCJAllowed @SCJRestricted(maySelfSuspend = false)
public abstract long size( )

The size of a memory area is always equal to the {memoryConsumed() + memoryRemaining()}.

returns the total size of this memory area.

MemoryArea.getMemoryArea(lValue).mayHoldReferenceTo();

MemoryArea.getMemoryArea(lValue).mayHoldReferenceTo(rValue);

7.3.5 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.

7.3.6 Class javax.realtime.ScopedMemory

Declaration

@SCJAllowed
public abstract class ScopedMemory
  implements javax.realtime.ScopedAllocationContext
  extends javax.realtime.MemoryArea

Description

Scoped memory implements the scoped allocation context. The only visible method is for resizing a scoped memory area.

7.3.7 Class javax.realtime.LTMemory

Declaration

@SCJAllowed
public class LTMemory **extends** javax.realtime.ScopedMemory

Description

This class can not be instantiated in SCJ. It is subclassed by MissionMemory and PrivateMemory. It has no visible methods.

7.3.8 Class javax.safetycritical.ManagedMemory

Declaration

@SCJAllowed
public abstract class ManagedMemory **extends** javax.realtime.LTMemory

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. Applications shall not directly extend this class.

Methods

@SCJAllowed
public static void enterPrivateMemory(**long** size, Runnable logic)
  throws java.lang.IllegalStateException

This method causes the calling schedulable object to execute the logic in a nested private memory area. If a private memory does not exist, one is created create with the specified size; otherwise, its size is set. Then the private memory area is cleared and entered.

The private memory object representing the inner scope memory area may be reused on subsequent calls to enterPrivateMemory during the lifetime of the current memory area.

size — is the size in bytes of the private memory.

logic — is the code to be executed in the private memory.

Throws IllegalStateException if the currently running thread is forbidden from entering a PrivateMemory area, such as when the current thread is executing within Mission.initialize or Safelet.getSequencer,

@SCJAllowed
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 that is allocated in the memory area that is entered.

logic — is the code to be executed in the entered memory area.

@SCJAllowed
public static void executeInOuterArea(Runnable logic)

Change the allocation context to the immediate outer memory area and invoke the run method if the Runnable.

logic — is the code to be executed in the entered memory area.

@SCJAllowed
public long getRemainingBackingStore( )

This method can be used to manage the use of backing store by any SCJ memory area.

returns the size of the remaining memory available to the current ManagedMemory area.

7.3.9 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.

SizeEstimator is a ceiling on the amount of memory that is consumed when the reserved objects are created.

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 (if, for example, the object and its monitor are separate), the size estimate shall include the sum of the sizes of all the 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.

Constructors

@SCJAllowed
@SCJRestricted(maySelfSuspend = false,
{javax.safetycritical.annotate.Phase.ALL})
public SizeEstimator( )

Creates a new SizeEstimator object in the current allocation context.

Methods

@SCJAllowed
@SCJRestricted(maySelfSuspend = false,
{javax.safetycritical.annotate.Phase.ALL})
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
@SCJRestricted(maySelfSuspend = false,
{javax.safetycritical.annotate.Phase.ALL})
public void reserve(SizeEstimator size, **int** num)

Take into account additional number instances of SizeEstimator when estimating the size.

size — is the given instance of SizeEstimator.

num — is the number of times to reserve the size denoted by size.

Throws IllegalArgumentException if size is null.

@SCJAllowed
@SCJRestricted(maySelfSuspend = false,
{javax.safetycritical.annotate.Phase.ALL})
public void reserve(SizeEstimator size)

Take into account an additional instance of SizeEstimator size when estimating the size.

size — is the given instance of SizeEstimator.

Throws IllegalArgumentException if size is null.

@SCJAllowed
@SCJRestricted(maySelfSuspend = false,
{javax.safetycritical.annotate.Phase.ALL})
public void reserve(**Class**<> clazz, **int** num)

Take into account additional num instances of Class clazz when estimating the size.

clazz — is the class to take into account.

num — is the number of instances of clazz to estimate.

Throws IllegalArgumentException if clazz is null.

```
@SCJAllowed
@SCJRestricted(maySelfSuspend = false,
  {javax.safetycritical.annotate.Phase.ALL})
public void reserveArray(int length, Class<> type)
```

Take into account an additional instance of an array of length primitive values. Class values for the primitive types are available from the corresponding class types; e.g., Byte.TYPE, Integer.TYPE, and Short.TYPE. The reservation will 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
@SCJRestricted(maySelfSuspend = false,
  {javax.safetycritical.annotate.Phase.ALL})
public void reserveArray(int length)
```

Take into account an additional instance of an array of length reference values.

length — is the number of entries in the array.

7.4 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. Traditionally, Java stores all objects in a heap that is subject to 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 not deallocated. Once allocated, an object is never reclaimed. 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. A single nesting structure is provided such that a given scoped memory can be entered only by a single thread at any given time and a scope may be entered only from the memory area in which it was created. These rules simplify scope entry analysis. Furthermore, although immortal memory is simple to understand, it has the limitation that the memory used by objects in immortal memory would not be reclaimable, even in a later mission. Therefore, each application uses a global mission scope (called mission memory) in the place of immortal memory to hold global objects used during a mission. The advantage is that all objects allocated in this mission memory can be reclaimed whenever the mission is restarted or replaced by another mission. Furthermore, it enables the avoidance of fragmentation in the underlying memory management system. This will enable confidence to be obtained with the use of dynamic memory, and for more expressive models to be developed in the future.

Corresponding to an assumed three-phase model of application execution, an SCJ system will allocate objects in mission memory in the initialization phase and then in private memory during the execution phase of the application. All class initialization happens before the initialization phase, and therefore with the immortal memory as the current allocation context. Class objects are allocated in immortal memory, as defined in the RTSJ, see Chapter 3.

7.4.1 Nesting Scopes

MissionMemory is just a ScopedMemory which is provided for the application during startup for holding objects that have a mission life span. This acts like an immortal memory area during a mission, except that it can 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 tasks have terminated.

Because the MissionMemory is not cleared during the mission, allocation of objects in the MissionMemory during the execution of the mission can lead to a memory leak; therefore, each schedulable object is given its own private scoped memory. Thus, the event handler classes available to the programmer are managed in the sense that each instance has its own PrivateMemory, that is entered on each release and exited at the end of each release. Because PrivateMemory is based on LTMemory, this gives a single level of nesting for the application.

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. A similar effect can be obtained in SCJ with the try statement that includes a finally clause for any given try scope.

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 done to aid application portability to other implementation.

7.5 Compatibility

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

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.

8.1 Semantics and Requirements

The resolution returned by a clock's getResolution() method is the resolution that shall be used for all scheduling decisions based on that clock.

The resolution and drift of the real-time clock is dependent on the underlying hardware clock and operating system implementation. Application developers should refer to the hardware clock specification as well as information from OS and SCJ vendors. See Section 4.7.4 for a discussion of the effects of clock granularity.

8.1.1 Clocks

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 monotonic and non-decreasing. The real-time clock in the RTSJ (queried through Clock.getRealtimeClock()) has an Epoch of January 1, 1970. In an SCJ system, the *Epoch* may represent the system start time. As a consequence, absolute times based on the real-time clock may not correspond to wall-clock time.

8.1.2 Time

Three time classes from the RTSJ are available for use in safety critical programs: AbsoluteTime, RelativeTime, and HighResolutionTime. As in the RTSJ, the base time class is HighResolutionTime. Both AbsoluteTime and RelativeTime are subclasses of HighResolutionTime. AbsoluteTime represents a specific point in time, while RelativeTime represents a time interval.

8.1.3 Application-defined Clocks

Every SCJ implementation shall provide a default real-time clock, but application developers are also able to define application-defined clocks. Such clocks can be referenced in the constructors of objects based on the AbsoluteTime and RelativeTime classes, and can therefore be used anywhere these objects are used. Application-defined clocks (and consequently timers that are based on those clocks) facilitate the release of periodic schedulable objects and timeouts based on application-detected events.

A application-defined clock need not be *monotonic* (in contrast to the default RTSJ real-time clock). As the application-defined clock is driven by application generated events, the notion of clock *resolution* and *uniformity* shall have an application-defined meaning.

8.1.4 RTSJ Constraints

Periodic, time-triggered application code is constructed using periodic event that trigger a (PeriodicEventHandler). The RTSJ classes OneShotTimer, PeriodicTimer, and Timer that can be used to schedule application logic in the RTSJ are not directly available in SCJ. For timeouts and non-periodic time-triggered release of a handler SCJ provides the OneShotEventHandler class.

Because java.util.Date is not part of the SCJ library, the related constructor in AbsoluteTime is omitted.

8.2 Level Considerations

Because wait and notify are available only in 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.

8.3 API

Figure 8.1 gives an overview of the time related classes.

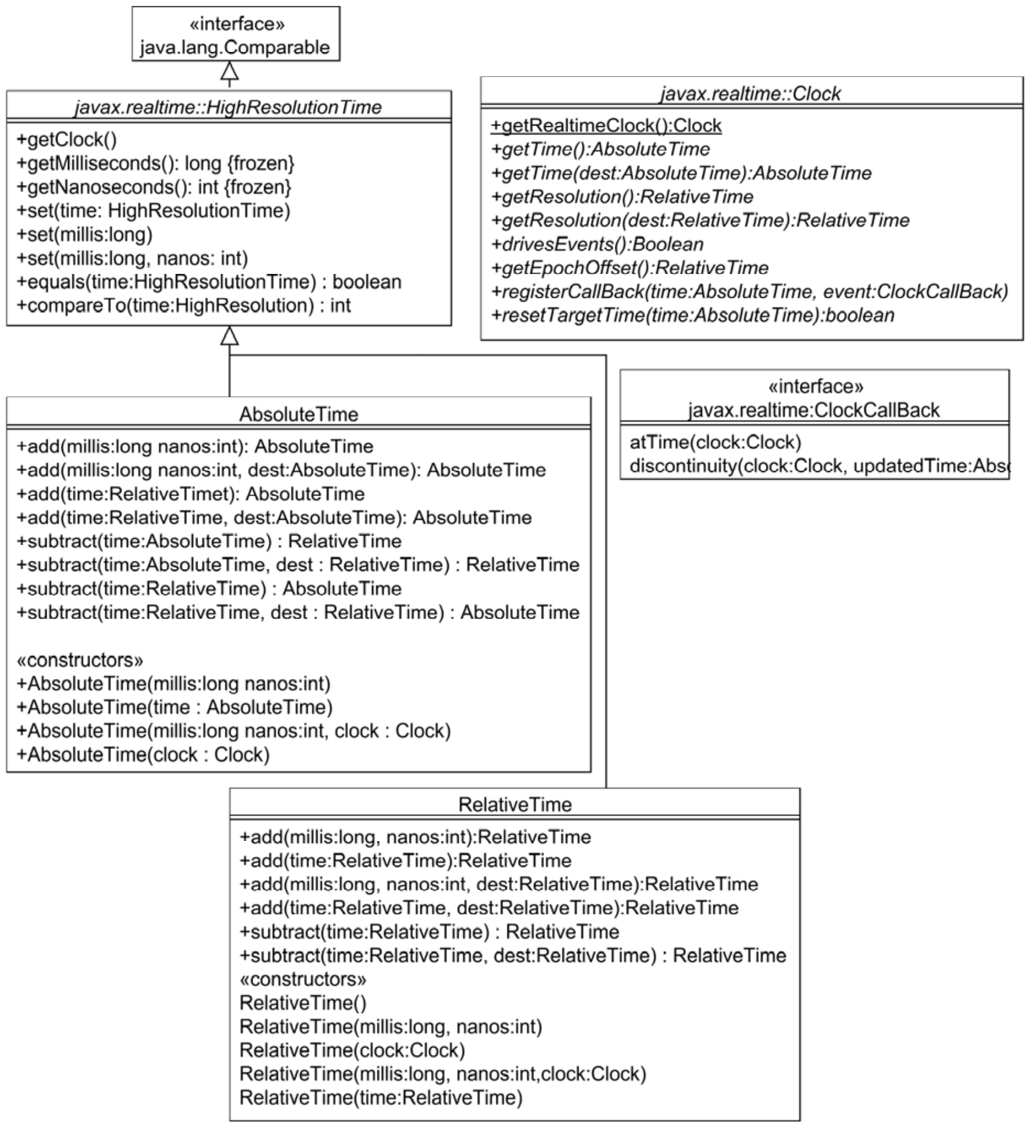
«interface»
java.lang.Comparable
javax.realtime::HighResolutionTime
+getClock()
+getMilliseconds(): long {frozen}
+getNanoseconds(): int {frozen}
+set(time: HighResolutionTime)
+set(millis:long)
+set(millis:long, nanos: int)
+equals(time:HighResolutionTime) : boolean
+compareTo(time:HighResolution) : int
javax.realtime::Clock
+getRealtimeClock():Clock
+getTime():AbsoluteTime
+getTime(dest:AbsoluteTime):AbsoluteTime
+getResolution():RelativeTime
+getResolution(dest:RelativeTime):RelativeTime
+drivesEvents():Boolean
+getEpochOffset():RelativeTime
+registerCallBack(time:AbsoluteTime, event:ClockCallBack)
+resetTargetTime(time:AbsoluteTime):boolean
«interface»
javax.realtime:ClockCallBack
atTime(clock:Clock)
discontinuity(clock:Clock, updatedTime:Abs
AbsoluteTime
+add(millis:long nanos:int): AbsoluteTime
+add(millis:long nanos:int, dest:AbsoluteTime): AbsoluteTime
+add(time:RelativeTimet): AbsoluteTime
+add(time:RelativeTime, dest:AbsoluteTime): AbsoluteTime
+subtract(time:AbsoluteTime) : RelativeTime
+subtract(time:AbsoluteTime, dest : RelativeTime) : RelativeTime
+subtract(time:RelativeTime) : AbsoluteTime
+subtract(time:RelativeTime, dest : RelativeTime) : AbsoluteTime
«constructors»
+AbsoluteTime(millis:long nanos:int)
+AbsoluteTime(time : AbsoluteTime)
+AbsoluteTime(millis:long nanos:int, clock : Clock)
+AbsoluteTime(clock : Clock)
RelativeTime
+add(millis:long, nanos:int):RelativeTime
+add(time:RelativeTime):RelativeTime
+add(millis:long, nanos:int, dest:RelativeTime):RelativeTime
+add(time:RelativeTime, dest:RelativeTime):RelativeTime
+subtract(time:RelativeTime) : RelativeTime
+subtract(time:RelativeTime, dest:RelativeTime) : RelativeTime
«constructors»
RelativeTime()
RelativeTime(millis:long, nanos:int)
RelativeTime(clock:Clock)
RelativeTime(millis:long, nanos:int,clock:Clock)
RelativeTime(time:RelativeTime)

Figure 8.1: Abridged time classes

8.3.1 Class javax.realtime.Clock

Declaration

@SCJAllowed
public abstract class Clock **extends** java.lang.Object

Description

A clock marks the passing of time. It has a concept of now that can be queried through Clock.getTime, and it can have events queued on it which will be fired 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 other clocks are not valid for any purpose that involves sleeping or waiting. They may, however, be used in the fire time and the period of OneShotTimer.

Constructors

@SCJAllowed @SCJRestricted(maySelfSuspend = false)
public Clock( )

Constructor for the abstract class.

Memory behavior: This constructor may allocate objects within the same MemoryArea that holds the implicit this argument.

Methods

@SCJAllowed @SCJRestricted(mayAllocate = false, maySelfSuspend = false)
protected abstract boolean drivesEvents( )

Returns true if and only if this Clock is able to trigger the execution of time-driven activities. Some user-defined clocks may be read-only, meaning the clock can be used to obtain timestamps, but the clock cannot be used to trigger the execution of events. If a clock that does not return drivesEvents equals true is used to configure a Timer or a sleep() request, an IllegalArgumentException will be thrown by the infrastructure. The default real-time clock does drive events.

returns true if the clock can drive events.

@SCJAllowed @SCJRestricted(maySelfSuspend = false)
public abstract javax.realtime.RelativeTime getEpochOffset( )

Returns the relative time of the offset of the epoch of this clock from the Epoch. For the real-time clock it will return a RelativeTime value equal to 0. An UnsupportedOperationException is thrown if the clock does not support the concept of date.

returns A newly allocated RelativeTime object in the current execution context with the offset past the Epoch for this clock. The returned object is associated with this clock.

Memory behavior: This constructor may allocate objects within the currently active MemoryArea.

@SCJAllowed @SCJRestricted(maySelfSuspend = false, mayAllocate = false)
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 @SCJRestricted(maySelfSuspend = false)
public abstract javax.realtime.RelativeTime getResolution( )

Gets the resolution of the clock, the nominal interval between ticks.

returns A newly allocated RelativeTime object in the current execution context representing the clock resolution. The returned object is associated with this clock.

@SCJAllowed @SCJRestricted(maySelfSuspend = false)
public abstract javax.realtime.RelativeTime getResolution(
  RelativeTime dest)

Gets the resolution of the clock, the nominal interval between ticks. The return value shall be associated with this clock. All relative time differences measured by this clock are approximately an integral multiple of the resolution.

dest — Return the relative time value in dest. If dest is null, a newly allocated RelativeTime object in the current execution context is returned. The returned object is associated with this clock.

returns dest is set to values representing the resolution of this. The returned object is associated with this clock.

```
@SCJAllowed @SCJRestricted(mayAllocate = false, maySelfSuspend = false)
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 an 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 which will be updated in place. The clock association of the dest parameter is ignored. 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 parameter, representing the current time, associated with this clock, or null if dest was null.

```
@SCJAllowed @SCJRestricted(maySelfSuspend = false)
public abstract 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.

Memory behavior: This constructor may allocate objects within the currently active MemoryArea.

```
@SCJAllowed(javax.safetycritical.annotate.Level.LEVEL_1)
@SCJRestricted(maySelfSuspend = false)
protected abstract void registerCallBack(AbsoluteTime time,
  ClockCallBack clockEvent)
```

Code in the abstract base Clock class makes this call to the subclass. The method is expected to implement a mechanism that will invoke atTime in ClockCallBack at time time, and if this clock is subject to discontinuities, invoke ClockCallBack.discontinuity(javax.realtime.Clock, javax.realtime.RelativeTime) each time a clock discontinuity is detected. registerCallBack of this clock and invocations of atTime and resetTargetTime of clockEvent are protected by a clock specific lock.

time — The absolute time value on this clock at which ClockCallBack.atTime(Clock) should be invoked.

clockEvent — The object that should be notified at time. If clockEvent is null, unregister the current clock event.

@SCJAllowed(javax.safetycritical.annotate.Level.LEVEL_1)
@SCJRestricted(maySelfSuspend = false)
protected abstract boolean resetTargetTime(AbsoluteTime time)

Replace the target time being used by the ClockCallBack registered by registerCallBack(AbsoluteTime, ClockCallBack).

time — The new target time.

returns false if no ClockEvent is currently registered.

8.3.2 Interface javax.realtime.ClockCallBack

Declaration

@SCJAllowed(javax.safetycritical.annotate.Level.LEVEL_1)
public interface ClockCallBack

Description

The ClockEvent interface may be used by subclasses of Clock to indicate to the clock infrastructure that the clock has either reached a designated time, or has experienced a discontinuity. Invocations of the methods in ClockCallBack are serialized.

Methods

@SCJAllowed(javax.safetycritical.annotate.Level.LEVEL_1)
@SCJRestricted(maySelfSuspend = false)
public void atTime(Clock clock)

Clock has reached the designated time. This clock event is de-registered before this method is invoked.

clock — the clock that has reached a designated time.

```
@SCJAllowed(javax.safetycritical.annotate.Level.LEVEL_1)
@SCJRestricted(maySelfSuspend = false)
public void discontinuity(Clock clock, AbsoluteTime updatedTime)
```

The clock experienced a time discontinuity (it changed its time value other than by ticking.) The clock has de-registered this clock event.

clock — the clock that has experienced a discontinuity.

updatedTime — the signed length of the time discontinuity.

8.3.3 Class javax.realtime.HighResolutionTime

Declaration

```
@SCJAllowed
public abstract class HighResolutionTime
  implements java.lang.Comparable>
  extends java.lang.Object
```

Description

Class HighResolutionTime is the base class for AbsoluteTime, and RelativeTime. Time can be expressed with nanosecond accuracy. This class is never used directly: it is abstract and has no public constructor. Instead, one of its subclasses AbsoluteTime or RelativeTime should be used.

Methods

```
@SCJAllowed @SCJRestricted(maySelfSuspend = false)
public int compareTo(HighResolutionTime time)
```

Compares this HighResolutionTime with the specified HighResolutionTimetime.

time — compares with the time of this.

```
@SCJAllowed @SCJRestricted(maySelfSuspend = false)
public int compareTo(Object object)
```

Compares this HighResolutionTime with the specified object.

object — compares with the time of this.

@SCJAllowed @SCJRestricted(maySelfSuspend = false, mayAllocate = false)
public boolean equals(HighResolutionTime time)

Returns true if the argument object has the same type and values as this.

time — Value compared to this.

returns true if the parameter object is of the same type and has the same values as this.

@SCJAllowed @SCJRestricted(maySelfSuspend = false, mayAllocate = false)
public boolean equals(Object object)

Returns true if the argument object has the same type and values as this.

object — Value compared to this.

returns true if the parameter object is of the same type and has the same values as this.

@SCJAllowed @SCJRestricted(maySelfSuspend = false)
public javax.realtime.Clock getClock( )

returns A reference to the clock associated with this.

@SCJAllowed @SCJRestricted(maySelfSuspend = false)
public final long getMilliseconds( )

returns The milliseconds component of the time represented by this.

@SCJAllowed @SCJRestricted(maySelfSuspend = false)
public final int getNanoseconds( )

returns The nanoseconds component of the time represented by this.

@SCJAllowed @SCJRestricted(maySelfSuspend = false, mayAllocate = false)
public int hashCode( )

Returns a hash code for this object in accordance with the general contract of Object.hashCode.

returns The hashcode value for this instance.

@SCJAllowed @SCJRestricted(maySelfSuspend = false)
public void set(**long** millis)

Sets the millisecond component of this to the given argument, and the nanosecond component of this to 0.

millis — This value shall be the value of the millisecond component of this at the completion of the call.

@SCJAllowed @SCJRestricted(maySelfSuspend = false, mayAllocate = false)
public void set(**long** millis, **int** nanos)

Sets the millisecond and nanosecond components of this.

millis — The desired value for the millisecond component of this at the completion of the call. The actual value is the result of parameter normalization.

nanos — The desired value for the nanosecond component of this at the completion of the call. The actual value is the result of parameter normalization.

@SCJAllowed @SCJRestricted(maySelfSuspend = false)
public void set(HighResolutionTime time)

Change the value represented by this to that of the given time.

time — The new value for this.

@SCJAllowed(javax.safetycritical.annotate.Level.LEVEL_2)
public static void 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.

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.

8.3.4 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 the 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. 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.

Constructors

```
@SCJAllowed
@SCJRestricted({javax.safetycritical.annotate.Phase.ALL},
   maySelfSuspend = false,
   mayAllocate = true)
public AbsoluteTime(long millis, int nanos)
```

Construct an AbsoluteTime object with time millisecond and nanosecond components past the real-time clock's Epoch.

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.

```
@SCJAllowed
@SCJRestricted({javax.safetycritical.annotate.Phase.ALL},
   maySelfSuspend = false,
   mayAllocate = true)
public AbsoluteTime( )
```

Equivalent to new AbsoluteTime(0,0).

```
@SCJAllowed
@SCJRestricted({javax.safetycritical.annotate.Phase.ALL},
   maySelfSuspend = false,
   mayAllocate = true)
public AbsoluteTime(AbsoluteTime time)
```

Make a new AbsoluteTime object from the given AbsoluteTime object.

time — AbsoluteTime object which is the source for the copy.

Memory behavior: This constructor requires that the "time.getClock()" argument reside in a scope that encloses the scope of the "this" argument.

```
@SCJAllowed
@SCJRestricted({javax.safetycritical.annotate.Phase.ALL},
   maySelfSuspend = false,
   mayAllocate = true)
public AbsoluteTime(long millis, int nanos, Clock clock)
```

Construct an AbsoluteTime object with time millisecond and nanosecond components past the epoch for clock.

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 clock providing the association for the newly constructed object.

```
@SCJAllowed
@SCJRestricted({javax.safetycritical.annotate.Phase.ALL},
   maySelfSuspend = false,
   mayAllocate = true)
public AbsoluteTime(Clock clock)
```

Equivalent to new AbsoluteTime(0,0,clock).

clock — The clock providing the association for the newly constructed object.

Methods

```
@SCJAllowed
@SCJRestricted({javax.safetycritical.annotate.Phase.ALL},
   maySelfSuspend = false,
   mayAllocate = true)
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.

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 result of the normalization of this plus millis and nanos in dest if dest is not null, otherwise the result is returned in a newly allocated object.

```
@SCJAllowed
@SCJRestricted({javax.safetycritical.annotate.Phase.ALL},
   maySelfSuspend = false,
   mayAllocate = true)
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.

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 result of the normalization of this plus the RelativeTime parameter time in dest if dest is not null, otherwise the result is returned in a newly allocated object.

```
@SCJAllowed
@SCJRestricted({javax.safetycritical.annotate.Phase.ALL},
   maySelfSuspend = false,
   mayAllocate = true)
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.

time — The time to add to this.

returns A new AbsoluteTime object whose time is the normalization of this plus the parameter time.

Memory behavior: This constructor may allocate objects within the currently active MemoryArea.

```
@SCJAllowed
@SCJRestricted({javax.safetycritical.annotate.Phase.ALL},
   maySelfSuspend = false,
   mayAllocate = true)
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.

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.

Memory behavior: This constructor may allocate objects within the currently active MemoryArea.

```
@SCJAllowed
@SCJRestricted({javax.safetycritical.annotate.Phase.ALL},
    maySelfSuspend = false,
    mayAllocate = true)
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.

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 result of the normalization of this minus the AbsoluteTime parameter time in dest if dest is not null, otherwise the result is returned in a newly allocated object.

```
@SCJAllowed
@SCJRestricted({javax.safetycritical.annotate.Phase.ALL},
    maySelfSuspend = false,
    mayAllocate = true)
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 result of the normalization of this minus the RelativeTime parameter time in dest if dest is not null, otherwise the result is returned in a newly allocated object.

```
@SCJAllowed
@SCJRestricted({javax.safetycritical.annotate.Phase.ALL},
    maySelfSuspend = false,
    mayAllocate = true)
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.

time — The time to subtract from this.

returns A new AbsoluteTime object whose time is the normalization of this minus the parameter time.

Memory behavior: This constructor may allocate objects within the currently active MemoryArea.

```
@SCJAllowed
@SCJRestricted({javax.safetycritical.annotate.Phase.ALL},
    maySelfSuspend = false,
    mayAllocate = true)
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.

time — The time to subtract from this.

returns A new RelativeTime object whose time is the normalization of this minus the AbsoluteTime parameter time.

Memory behavior: This constructor may allocate objects within the currently active MemoryArea.

8.3.5 Class javax.realtime.RelativeTime

Declaration

@SCJAllowed
public class RelativeTime **extends** javax.realtime.HighResolutionTime

Description

An object that represents a time interval milliseconds/10$\hat{3}$ + nanoseconds/10$\hat{9}$ seconds long that is divided into subintervals by some frequency. This is generally used in periodic events, threads, and feasibility analysis to specify periods where there is a basic period that must be adhered to strictly (the interval), but within that interval the periodic events are supposed to happen frequency times, as uniformly spaced as possible, but clock and scheduling jitter is moderately acceptable.

Constructors

@SCJAllowed @SCJRestricted(maySelfSuspend = false)
public RelativeTime( )

Equivalent to new RelativeTime(0,0).

@SCJAllowed @SCJRestricted(maySelfSuspend = false)
public RelativeTime(**long** ms, **int** ns)

Construct a RelativeTime object representing an interval based on the parameter millis plus the parameter nanos. Todo: say which clock this RelativeTime is associated with.

ms — The desired value for the millisecond component of this. The actual value is the result of parameter normalization.

ns — The desired value for the nanosecond component of this. The actual value is the result of parameter normalization.

@SCJAllowed @SCJRestricted(maySelfSuspend = false)
public RelativeTime(Clock clock)

Equivalent to new RelativeTime(0,0,clock).

clock — The clock providing the association for the newly constructed object.

@SCJAllowed @SCJRestricted(maySelfSuspend = false)
public RelativeTime(**long** ms, **int** ns, Clock clock)

Construct a RelativeTime object representing an interval based on the parameter millis plus the parameter nanos.

ms — The desired value for the millisecond component of this. The actual value is the result of parameter normalization.

ns — The desired value for the nanosecond component of this. The actual value is the result of parameter normalization.

clock — The clock providing the association for the newly constructed object.

```
@SCJAllowed @SCJRestricted(maySelfSuspend = false)
public RelativeTime(RelativeTime time)
```

Make a new RelativeTime object from the given RelativeTime object.

time — The RelativeTime object which is the source for the copy.

Methods

```
@SCJAllowed @SCJRestricted(maySelfSuspend = false)
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.

time — The time to add to this.

returns A new RelativeTime object whose time is the normalization of this plus millis and nanos.

Memory behavior: This constructor may allocate objects within the currently active MemoryArea.

```
@SCJAllowed @SCJRestricted(maySelfSuspend = false)
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.

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 result of the normalization of this plus the RelativeTime parameter time in dest if dest is not null, otherwise the result is returned in a newly allocated object.

```
@SCJAllowed @SCJRestricted(maySelfSuspend = false)
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.

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 result of the normalization of this plus millis and nanos in dest if dest is not null, otherwise the result is returned in a newly allocated object.

```
@SCJAllowed @SCJRestricted(maySelfSuspend = false)
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.

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.

Memory behavior: This constructor may allocate objects within the currently active MemoryArea.

```
@SCJAllowed @SCJRestricted(maySelfSuspend = false)
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.

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 result of the normalization of this minus the RelativeTime parameter time in dest if dest is not null, otherwise the result is returned in a newly allocated object.

@SCJAllowed @SCJRestricted(maySelfSuspend = false)
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.

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.

Memory behavior: This constructor may allocate objects within the currently active MemoryArea.

8.4 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 time since system startup time.

As time values from different clocks 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.

Application-defined clocks result in a tight interaction between the infrastructure and the application. Figure 8.2 shows a sequence diagram that illustrates the usage of a application-defined clock.

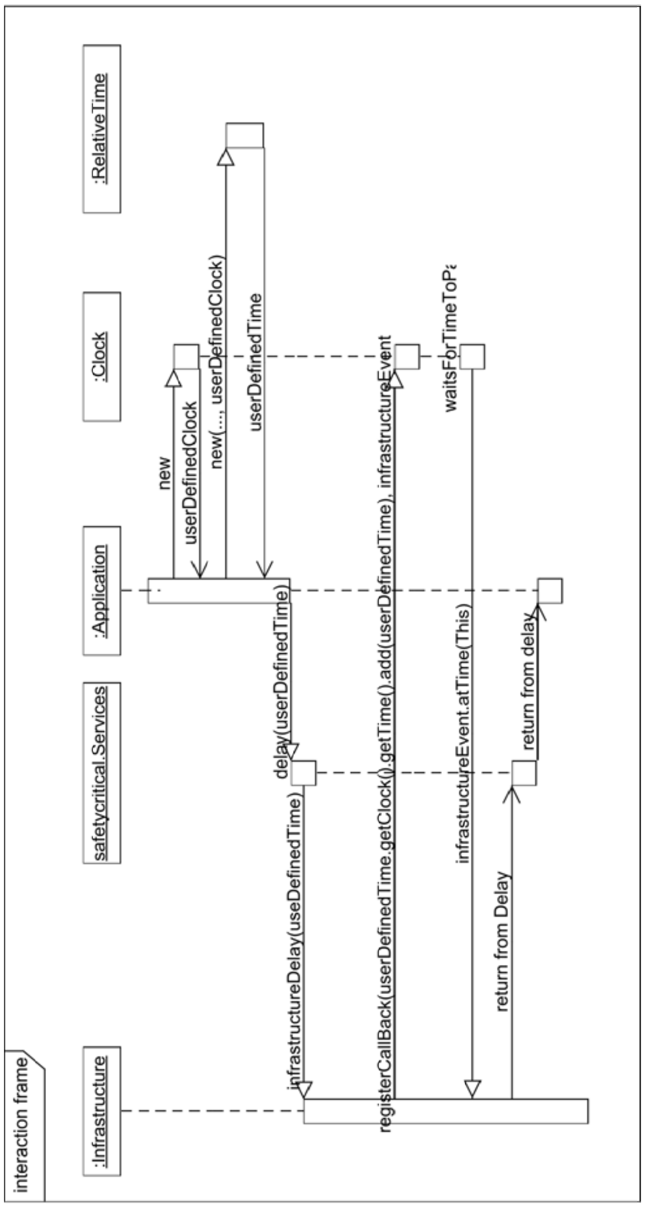

Figure 8.2: Sequence diagram of an application-defined clock

8.5 Compatibility

The RTSJ does not permit using application-defined clocks for scheduling. SCJ permits base scheduling decisions to be based on application-defined clocks.

The RTSJ defines the Epoch to be the 1st January 1970.

Chapter 9

Java Metadata Annotations

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. These metadata annotations provide a basis for additional checks for ensuring the correctness and efficiency of safety-critical Java programs. They are retained in the compiled bytecode intermediate format and are thus available for performing validation at class load-time. One strong SCJ interest in using metadata annotations is to ensure memory safety, thus preventing several exceptions from being thrown at runtime. They are also used for enforcement of compliance levels and restricting the behavior of certain methods.

This specification differentiates between *application code* and *infrastructure code*. Application code is checked by the tool to abide by the restrictions outlined in this chapter. Infrastructure code is verified by the vendor. Infrastructure code includes the java and javax packages as well as vendor specific libraries.

A *Checker* program is provided along with the SCJ Reference Implementation, and the Technology Compatibility Kit that can be used to analyze an SCJ-compliant application program to statically check the properties addressed by the annotations in this chapter.

9.1 Semantics and Requirements

The SCJ annotations address the following three 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, a code belonging to a certain level may access only the code that is at the same or higher level. This ensures that an SCJ application is consistent with respect to the specified SCJ level.

- *Behavioral Restrictions*—Because the execution of the missions is implemented as a sequence of specific phases (initialization, execution, cleanup), the application must clearly distinguish between these phases. Furthermore, it is illegal to access SCJ functionality that is not provided for the current execution phase of a mission.
- *Memory Safety*—Memory safety is a very important aspect of safety-critical software, but it is not currently supported in this SCJ specification. It is expected that implementations will provide support for annotations designed to ensure memory safety. This topic is further discussed in the Rationale section of this chapter.

9.2 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. Because the SCJ specification spans more package names (e.g. javax.realtime and javax.safetycritical), package-private visibility is not an option.

The SCJ specification specifies three compliance levels to which applications and implementations may conform. Each level specifies restrictions on what APIs may be used, with lower levels being 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**	Application-level.
		LEVEL_1	
		LEVEL_2	
		SUPPORT	Application-level, accessed by library.
		INFRASTRUCTURE	Library private.
		HIDDEN	Non-accessible.
	members	TRUE	Inherit value by sub-elements.
		FALSE	

Table 9.1: Compliance LEVEL annotation. Default values in bold.

1. The default argument of type Level specifies the level of the annotation target. The options are LEVEL_0, LEVEL_1, LEVEL_2, SUPPORT, INFRASTRUCTURE and HIDDEN.

- 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 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.
- INFRASTRUCTURE specifies that a method is API private. Therefore, methods outside of javax.realtime and javax.safetycritical packages may not invoke methods that have this annotation.
- HIDDEN denotes classes and methods that are hidden and can not be accessed both from the application and infrastructure code. No element with this annotation can be accessed from the SCJ application or infrastructure.

The default value when no value is specified is LEVEL_0. When no annotation applies to a class or member, it takes on value HIDDEN. The ordering on annotations is LEVEL_0 < LEVEL_1 < LEVEL_2 < SUPPORT <INFRASTRUCTURE < HIDDEN.

2. The second argument, members, determines whether or not the specified compliance level recurses to nested members and classes. The default value is false.

9.2.0.1 Overriding the Default Level for Application Classes

By default, any infrastructure and application code has the level set to HIDDEN. The default value for the application-level code can be overriden by a command-line argument -Alevel= passed to the Checker. The possible values of the argument are -Alevel=0, -Alevel=1, and -Alevel=2, corresponding to LEVEL_0, LEVEL_1, and LEVEL_2 respectively.

9.2.0.2 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(a)nnotation, if it exists,

2. The level of the closest outer element with an @SCJAllowed(a)nnotation, if members = true,

3. HIDDEN.

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 of enclosed elements of a class or member shall have a compliance level greater than or equal to the enclosing element.

Methods annotated HIDDEN or INFRASTRUCTURE may not be overridden in application code. 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.

9.2.0.3 Class Constructor Rules

For a class that is annotated @SCJAllowed, all constructors shall be annotated @SCJAllowed as well.

If a class has a default constructor, the constructor's compliance level shall be that of the class if the annotation has members = true, or HIDDEN otherwise.

9.2.0.4 Other Rules

The exceptions thrown by a method must be visible at the compliance level of that method.

9.3 Annotations for Restricting Behavior

The following set of annotations is provided to express behaviors and characteristics of methods. For example, some methods may only be called in a certain mission phase. Others may be restricted from allocation or blocking calls. In both cases, the restricted behavior annotation @SCJRestricted is used.

The SCJRestricted annotation has three attributes: mayAllocate, maySelfSuspend, and value. The first two are boolean and the last takes an element of the Phase enumeration.

When mayAllocate is true, the annotated method is allowed to perform allocation or call methods that are also annotated @SCJRestricted(mayAllocate = true). If a method is @SCJRestricted(mayAllocate = false), then all method that override it must be @SCJRestricted(mayAllocate = false) as well. Only methods that are annotated @SCJRestricted(mayAllocate = true) may contain expressions that result in allocation (e.g. at the source level new expressions, string concatenation, and auto-boxing). The default value is true.

When maySelfSuspend is true, the annotated method may take an action that causes it to block. If a method is marked @SCJRestricted(maySelfSuspend = false, then neither it nor any method it calls may take an action causing it to block. The default value is true.

If a method annotated with value is set to anything other than ALL in SCJRestricted, then the method may only be called in the given phase.

The @SCJRestricted annotation may be set on a class, interface, or enumeration, in which case it changes the default values for the methods on that class, interface, or enumeration.

9.4 Level Considerations

These annotations apply to all levels.

9.5 API

9.5.1 Class javax.safetycritical.annotate.SCJRestricted

Declaration

```
@Retention(CLASS)
@Target( { TYPE, FIELD, METHOD, CONSTRUCTOR })
public @interface SCJRestricted
```

Methods

```
public Phase[] value() default {ANY_TIME}
public boolean mayAllocate() default true
public boolean maySelfSuspend() default false;
```

Description

This annotation distinguishes methods that may be called only from a certain context (e.g. CleanUp) or methods that may be restricted to execute no memory allocation or blocking.

9.5.2 Class javax.safetycritical.annotate.SCJAllowed

Declaration

```
@Retention(CLASS)
@Target( { TYPE, FIELD, METHOD, 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 this 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.

Methods

```
public Level value() default LEVEL_0
```

9.5.3 Class javax.safetycritical.annotate.Level

Declaration

```
public enum Level

  LEVEL_0
  LEVEL_1
  LEVEL_2
  SUPPORT
  INFRASTRUCTURE
  HIDDEN
```

Description

Provides a set of possible values for the @SCJAllowed annotation's argument *level*.

9.5.4 Class javax.safetycritical.annotate.Phase

Declaration

```
public enum Phase

  INITIALIZATION
  RUN
  CLEANUP
  ALL
```

Description

Provides a set of possible values for the @SCJRestricted annotation value.

9.6 Rationale and Examples

It is expected that the metadata annotations will be checked at compile time as well as at load time (or link time if class loading is integrated with the linking). Compile-time checking is useful to provide rapid feedback to developers, while load or link time checking is essential for ensuring safety. 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.

9.6.1 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
    peh.run();                 // ERROR
   }
}

@SCJAllowed (LEVEL_0)
public interface Schedulable extends SCJRunnable {

 @SCJAllowed(LEVEL_2)
 public ReleaseParameters getReleaseParameters();
}

@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
  public ReleaseParameters getReleaseParameters() {...}

  @SCJAllowed(INFRASTRUCTURE) public final void run() {...}
}
```

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. Furthermore, MyMission's initialization method attempts to instantiate a MyHandler object and consequently tries to execute its functionality by calling PeriodicEventHandler's run() method. However, the method is annotated as @SCJAllowed(INFRASTRUCTURE), which indicates that it can be called only from the SCJ infrastructure code.

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 getReleaseParameters(), originally allowed only at Level 2. This results in an illegal attempt to decrease method visibility.

9.6.2 Memory Safety Annotations

In an earlier draft of this SCJ specification, a set of annotations designed to ensure the safety of memory references were included in this Chapter. Because the SCJ Expert Group determined that those proposed memory safety annotations were not ready for standardization, they were moved to an Appendix (see Appendix H).

Chapter 10

JNI

10.1 Semantics and Requirements

The RTSJ provides only minimal restrictions on calls to native interfaces. SCJ provides more restrictions than the RTSJ to simplify the run-time infrastructure and assist with safety-critical analysis. This chapter defines these additional SCJ restrictions. If the underlying run-time infrastructure supports native code execution, then all JNI supported services described in this chapter shall be implemented; otherwise, JNI is not available to the application.

10.2 Level Considerations

Due to SCJ limitations concerning reflection and object allocation the JNI support is constricted to a basic fuctionality. The remaining services can be used equally for native methods on Level 0, Level 1, and Level 2.

10.3 API

10.3.1 Supported Services

All of the JNI services in this section are supported by SCJ implementations that support JNI.

General service to get JNI version information:

- GetVersion

General object analysis: The following methods provide basic operations on objects and require no reflection, no object allocation, or other hard-to-analyze code.

- GetObjectClass
- IsInstanceOf
- IsSameObject
- GetSuperclass
- IsAssignableFrom

String Functions: The following methods provide basic operations on strings and require no reflection or other hard-to-analyze code.

- GetStringLength
- GetStringUTFLength
- GetStringRegion
- GetStringUTFRegion

Array Operations The following methods provide basic operations on arrays and require no reflection or other hard-to-analyze code.

- GetArrayLength
- GetObjectArrayElement
- SetObjectArrayElement
- Get $<$ PrimitiveType $>$ ArrayRegion routines
- Set $<$ PrimitiveType $>$ ArrayRegion routines

Native Function Registering: The following function is required to be supported, though it shall be called only during initialization. This function is needed to disambiguate between the two possible naming conventions for JNI functions, in systems where the Java implementation does not control linking.

- RegisterNatives

The following functions are required to be supported because they are easy to implement, and provide better compatibility with existing JNI code:

- DeleteLocalRef
- EnsureLocalCapacity
- PushLocalFrame
- PopLocalFrame
- NewLocalRef

10.3.2 Annotations

There is no SCJ support to verify the annotations of native methods. On the other hand it is important to provide this information to the tools validating SCJ programs for correctness and further purposes. To ensure that the application programmers consider their implementation carefully, there are no default annotations for native methods concerning allocation and blocking. Therefore it is always required to decorate native methods with either @MAY_BLOCK or @BLOCK_FREE. The same applies to @MAY_ALLOCATE and @ALLOCATE_FREE. @MAY_ALLOCATE indicates that the native method allocates native memory dynamically. SCJ compliant implementations of native methods cannot allocate objects in SCJ memory.

Note that any JNI code that may be called within a synchronized method shall be annotated as @BLOCK_FREE to indicate that it will never self-suspend.

As usual for SCJ source code, an annotation with @SCJAllowed() is also required for each native method.

10.4 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 services that would normally be available in JNI are not supported. In addition, no services that require allocation will be required for SCJ conformance.

Call-back services from C to create, attach or unload the JVM are not required because the corresponding operations are not supported.

10.4.1 Unsupported Services

These VM-related invocation API functions are not required to be supported:

- JNI_GetDefaultJavaVMInitArgs
- JNI_GetCreatedJavaVMs
- JNI_CreateJavaVM
- JNI_DestroyJavaVM
- JNI_AttachCurrentThread
- JNI_AttachCurrentThreadAsDaemon
- JNI_DetachCurrentThread
- JNI_GetEnv

There is no support for the native interface definitions to be redefined with the JNI OnLoad and JNI OnUnload services.

Primitive types, objects and arrays can all be passed into the underlying C function from Java using JNI.

The following methods are NOT required to be supported because they require reflection:

- NewObject
- NewObjectA
- NewObjectV
- GetFieldID
- Get < type > Field
- Set < type > Field
- GetStaticFieldID
- GetStatic < type > Field
- SetStatic < type > Field
- GetMethodID
- Call < type > Method
- Call < type > MethodA
- Call < type > MethodV
- GetStaticMethodID
- CallStatic < type > Method
- CallStatic < type > MethodA
- CallStatic < type > MethodV
- CallNonvirtual < type > Method
- CallNonvirtual < type > MethodA
- CallNonvirtual < type > MethodV
- FromReflectedMethod
- FromReflectedField
- ToReflectedMethod
- ToReflectedField

The following methods are not supported because they require allocation:

- NewString
- NewStringUTF
- NewObjectArray
- NewDirectByteBuffer
- GetStringChars
- GetStringUTFChars
- ReleaseStringChars
- ReleaseStringUTFChars
- New < type > Array
- Get < type > ArrayElements
- Release < type > ArrayElements
- GetStringCritical
- Release StringCritical

- GetPrimitiveArrayCritical
- ReleasePrimitiveArrayCritical

The following function is NOT required to be supported because it is only useful for systems with dynamic loading:

- UnregisterNatives

The following memory management services 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
- NewWeakGlobalRef
- DeleteWeakGlobalRef
- NewGlobalRef
- DeleteGlobalRef
- DeleteLocalRef

The following methods are NOT required to be supported because they map to 'synchronized' which is restricted:

- MonitorEnter
- MonitorExit

The following methods are NOT required to be supported because they require reflection and/or dynamic class loading to operate:

- DefineClass
- FindClass

10.5 Example

```
@SCJAllowed
@ALLOCATE_FREE
@MAY_BLOCK
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 @ALLOCATE_FREE, the implementation of getProcessorId must not allocate memory dynamically. Because the desired information might be obtained by a call to the operation system, @MAY_BLOCK is used.

Header files of the native implementation can be generated by javah as usual. The native implementation follows the common JNI rules found at **http://docs.oracle.com/javase/6/docs/technote** obeying the restrictions of the previous section.

10.6 Compatibility

10.6.1 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 services that are not supported.

For consistency with standard RTSJ applications, if an SCJ implementation supports facultative allocation of Java objects from native code, such allocations should allocate objects using the current allocation context at the point of the call.

10.6.2 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 services 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

Exceptions are normally considered a good mechanism to separate functional logic from error handling. Safety-critical applications in languages such as Ada and C++, however, usually avoid their use. One reason is that the possibility of exception propagation introduces run-time paths which are complicated to analyze.

In Java, it is typically impossible to avoid exception handlers altogether due to checked exceptions which can be thrown by many standard methods. Compiler analysis such as dataflow analysis can go a long way toward ensuring that certain exceptions will never be thrown by a given method invocation, but in general it is not possible to eliminate all throw statements and codecatch clauses.

This chapter describes how exceptions can be thrown and caught within SCJ programs without any risk of memory leaks, out-of-memory exceptions, or scope related exceptions. Observing these rules permits safe exception handling which may also be employed within application classes.

In this chapter the term *exception* may refer to any Throwable.

11.1 Semantics and Requirements

There are no special requirements on the allocation of exception objects. Exception object allocation through the keyword new uses the current allocation context; exceptions can be allocated in other allocation contexts by using that memory area's newInstance methods.

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.

In SCJ, every Schedulable shall be configured at construction time to set aside a thread-local buffer to represent stack back trace information associated with the exception most recently thrown by this Schedulable. See StorageParameters in Chapter 4.

It is implementation-defined how a particular implementation of SCJ captures and represents thread backtraces for thrown exceptions. See the Rationale section of this Chapter for a description of one possible approach.

11.1.1 SCJ-Specific Functionality

A ThrowBoundaryError exception which is thrown due to a boundary error 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 back-trace information is copied at this same time is implementation-defined.

The method getPropagatedExceptionClass() returns a reference to the class of the original exception. The method getPropagatedMessage returns the message associated with the original exception. 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 array 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.

The RTSJ defines a number of exceptions, thrown at run-time, when assignment rules between memory areas are violated. To avoid these exceptions, Chapter 9 introduces annotations for scope safe SCJ programs. Correctly annotated programs are guaranteed never to throw any of the scope related exceptions (IllegalAccessException, ScopedCycleException, InaccessibleAreaException).

11.2 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.

11.3 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.

11.3.1 Class java.lang.Error

Declaration

@SCJAllowed
public class Error
  implements java.io.Serializable
  extends java.lang.Throwable

Constructors

@SCJAllowed @SCJRestricted(maySelfSuspend = false)
public Error( )

> Constructs an Error object for an SCJ system with a null detail message. Invokes System.captureStackBacktrace(this) to save the back trace associated with the current thread.
>
> Allocates an application- and implementation-defined amount of memory in the current scope (to represent stack backtrace).
>
> This constructor shall not copy this to any instance or static field.

Memory behavior: This constructor may allocate objects within the currently active MemoryArea.

@SCJAllowed @SCJRestricted(maySelfSuspend = false)
public Error(String msg)

Constructs an Error object for an SCJ system with a detail message. Invokes System.captureStackBacktrace(this) to save the back trace associated with the current thread.

Allocates an application- and implementation-defined amount of memory in the current scope (to represent stack backtrace).

This constructor shall not copy this to any instance or static field.

msg — the detail message for this Error object.

Memory behavior: This constructor may allocate objects within the currently active MemoryArea.

This constructor requires that the "msg" argument reside in a scope that encloses the scope of the "this" argument.

@SCJAllowed @SCJRestricted(maySelfSuspend = false)
public Error(String msg, Throwable t)

Constructs an Error object for an SCJ system with a specified detail message and with a specified cause. Does not invoke System.captureStackBacktrace(this) to avoid overwriting the back trace associated with the cause.

Allocates an application- and implementation-defined amount of memory in the current scope (to represent stack backtrace).

This constructor shall not copy this to any instance or static field.

msg — the detail message for this Error object.

t — the exception that caused this error.

Memory behavior: This constructor may allocate objects within the currently active MemoryArea.

This constructor requires that the "msg" argument reside in a scope that encloses the scope of the "this" argument. This constructor requires that the "t" argument reside in a scope that encloses the scope of the "this" argument.

@SCJAllowed @SCJRestricted(maySelfSuspend = false)
public Error(Throwable t)

Constructs an Error object for an SCJ system with a null detail message and with a specified cause. Does not invoke System.captureStackBacktrace(this) to avoid overwriting the back trace associated with the cause.

Allocates an application- and implementation-defined amount of memory in the current scope (to represent stack backtrace).

This constructor shall not copy this to any instance or static field.

t — the exception that caused this error.

Memory behavior: This constructor may allocate objects within the currently active MemoryArea.

This constructor requires that the "t" argument reside in a scope that encloses the scope of the "this" argument.

11.3.2 Class java.lang.Exception

Declaration

@SCJAllowed
public class Exception
  implements java.io.Serializable
  extends java.lang.Throwable

Constructors

@SCJAllowed @SCJRestricted(maySelfSuspend = false)
public Exception( )

Constructs an Exception object for an SCJ system with a null detail message. Invokes System.captureStackBacktrace(this) to save the back trace associated with the current thread.

Allocates an application- and implementation-defined amount of memory in the current scope (to represent stack backtrace).

This constructor shall not copy this to any instance or static field.

Memory behavior: This constructor may allocate objects within the currently active MemoryArea.

@SCJAllowed @SCJRestricted(maySelfSuspend = false)
public Exception(String msg)

Constructs an Exception object for an SCJ system with a specified detail message. Invokes System.captureStackBacktrace(this) to save the back trace associated with the current thread.

This constructor shall not copy this to any instance or static field.

msg — the detail message for this Exception object.

Memory behavior: This constructor may allocate objects within the currently active MemoryArea.

This constructor requires that the "msg" argument reside in a scope that encloses the scope of the "this" argument.

@SCJAllowed @SCJRestricted(maySelfSuspend = false)
public Exception(Throwable cause)

Constructs an Exception object for an SCJ system with a null detail message and a specified cause. Does not invoke System.captureStackBacktrace(this) to avoid overwriting the back trace associated with the cause.

This constructor shall not copy this to any instance or static field.

cause — the cause of this exception.

Memory behavior: This constructor requires that the "cause" argument reside in a scope that encloses the scope of the "this" argument.

@SCJAllowed @SCJRestricted(maySelfSuspend = false)
public Exception(String msg, Throwable cause)

Constructs an Exception object for an SCJ system with a specified detail message and a specified cause. Does not invoke System.captureStackBacktrace(this) to avoid overwriting the back trace associated with the cause.

This constructor shall not copy this to any instance or static field.

msg — the detail message for this Exception object.

cause — the cause of this exception .

Memory behavior: This constructor requires that the "cause" argument reside in a scope that encloses the scope of the "this" argument. This constructor requires that the "msg" argument reside in a scope that encloses the scope of the "this" argument.

11.3.3 Class java.lang.Throwable

Declaration

```
@SCJAllowed
public class Throwable
  implements java.io.Serializable
  extends java.lang.Object
```

Constructors

```
@SCJAllowed
public Throwable( )
```

Constructs a Throwable object for an SCJ system with a null detail message and no specified cause. Invokes System.captureStackBacktrace(this) to save the back trace associated with the current thread.

This constructor shall not copy this to any instance or static field.

```
@SCJAllowed
@SCJRestricted({javax.safetycritical.annotate.Phase.ALL},
   maySelfSuspend = false,
   mayAllocate = true)
public Throwable(Throwable cause)
```

Constructs a Throwable object for an SCJ system with a null detail message and a specified cause. Does not invoke System.captureStackBacktrace(this) to avoid overwriting the back trace associated with the cause.

This constructor shall not copy this to any instance or static field.

cause — the cause of this exception.

Memory behavior: This constructor requires that the "cause" argument reside in a scope that encloses the scope of the "this" argument.

```
@SCJAllowed
@SCJRestricted({javax.safetycritical.annotate.Phase.ALL},
   maySelfSuspend = false,
   mayAllocate = true)
public Throwable(String msg, Throwable cause)
```

Constructs a Throwable object for an SCJ system with a specified detail message and a specified cause. Does not invoke System.captureStackBacktrace(this) to avoid overwriting the back trace associated with the cause.

This constructor shall not copy this to any instance or static field.

msg — the detail message for this Throwable object.

cause — the cause of this exception.

Memory behavior: This constructor requires that the "cause" argument reside in a scope that encloses the scope of the "this" argument. This constructor requires that the "msg" argument reside in a scope that encloses the scope of the "this" argument.

```
@SCJAllowed
@SCJRestricted({javax.safetycritical.annotate.Phase.ALL},
   maySelfSuspend = false,
   mayAllocate = true)
public Throwable(String msg)
```

Constructs a Throwable object for an SCJ system with a specified detail message and no specified cause. Invokes System.captureStackBacktrace(this) to save the back trace associated with the current thread.

This constructor shall not copy this to any instance or static field.

msg — the detail message for this Throwable object.

Memory behavior: This constructor requires that the "msg" argument reside in a scope that encloses the scope of the "this" argument.

Methods

```
@SCJAllowed
@SCJRestricted({javax.safetycritical.annotate.Phase.ALL},
   maySelfSuspend = false,
   mayAllocate = true)
public java.lang.Throwable getCause( )
```

returns a reference to the same Throwable that was supplied as an argument to the constructor, or null if no cause was specified at construction time. Performs no memory allocation.

```
@SCJAllowed
@SCJRestricted({javax.safetycritical.annotate.Phase.ALL},
   maySelfSuspend = false,
   mayAllocate = true)
public java.lang.String getMessage( )
```

returns a reference to the same String 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
@SCJRestricted({javax.safetycritical.annotate.Phase.ALL},
   maySelfSuspend = false,
   mayAllocate = true)
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 thread-local buffer to represent the stack back trace information associated with the most recent invocation of System.captureStackBacktrace . The size of this buffer is specified by providing a StorageParameters object as an argument to construction of the Schedulable. Most commonly, System.captureStackBacktrace is invoked from within the constructor of java.lang.Throwable . getStackTrace returns a representation of this thread-local back trace information.

If System.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 back trace for this particular Throwable. Shall not copy this to any instance or static field.

Memory behavior: This constructor may allocate objects within the currently active MemoryArea.

11.3.4 Class jaxax.safetycritical.ThrowBoundaryError

Declaration

@SCJAllowed
public class ThrowBoundaryError

extends javax.realtime.ThrowBoundaryError

Description

One ThrowBoundaryError is preallocated for each Schedulable in its outermost private scope.

Constructors

@SCJAllowed @SCJRestricted(maySelfSuspend = false)
public ThrowBoundaryError( )

Allocates an application- and implementation-defined amount of memory in the current scope (to represent the stack backtrace). Shall not copy "this" to any instance or static field.

Memory behavior: This constructor may allocate objects within the currently active MemoryArea.

Methods

@SCJAllowed @SCJRestricted(maySelfSuspend = false, mayAllocate = false)
public java.lang.**Class**> getPropagatedExceptionClass( )

returns a reference to the Class of the exception most recently thrown across a scope boundary by the current thread. Performs no allocation. Shall not copy this to any instance or static field.

@SCJAllowed @SCJRestricted(maySelfSuspend = false)
public java.lang.String getPropagatedMessage( )

returns a newly allocated String object and its backing store to represent the message associated with the thrown exception that most recently crossed a scope boundary within this thread.

The original message is truncated if it is longer than the length of the thread-local \texttt{StringBuilder} object, which length is specified in the \texttt{Storage\-Con\-fig\-ura\-tion\-Pa\-ra\-meters} for this \texttt{Schedulable}.

Shall not copy "this" to any instance or static field.

Memory behavior: This constructor may allocate objects within the currently active MemoryArea.

@SCJAllowed @SCJRestricted(maySelfSuspend = false)
public java.lang.StackTraceElement[] getPropagatedStackTrace( )

returns returns a newly allocated 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 ThrowBoundaryError object. Shall not copy "this" to any instance or static field.

Most commonly, System.captureStackBacktrace() is invoked from within the constructor of java.lang.Throwable. getPropagatedStackTrace() returns a representation of this thread-local back trace information.

Under normal circumstances, this stack back trace information corresponds to the exception represented by this ThrowBoundaryError object. However, certain execution sequences may overwrite the contents of the buffer so that the stack back trace information so that the stack back trace information is not relevant.

Memory behavior: This constructor may allocate objects within the currently active MemoryArea.

@SCJAllowed @SCJRestricted(maySelfSuspend = false, mayAllocate = false)
public int getPropagatedStackTraceDepth( )

returns the number of valid elements stored within the StackTraceElement array to be returned by getPropagatedStackTrace. Performs no allocation. Shall not copy this to any instance or static field. array to be returned by getPropagatedStackTrace().

11.4 Rationale

SCJ allows individual threads to set aside different buffer sizes for back trace information. During debugging, we expect that developers may want to set aside large buffers in order to maximize access to debugging information. However, during final deployment, many systems would run with minimal buffer sizes in order to reduce memory requirements and simplify the run-time behavior. Establishing the size of the stack back trace buffer at Schedulable construction time relieves the SCJ implementation from having to dynamically allocate memory when dealing with throw boundary errors.

The required 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 Services.captureBackTrace() to save the current thread's stack back trace into the thread-local buffer configured by this thread's StorageParameters.
- Services.captureBackTrace() takes a single Throwable argument which is the object with which to associate the back trace. 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 back trace 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 back trace 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 back trace buffer.
- In case application programs desire to throw preallocated exceptions, the application program has the option to invoke Services.captureBackTrace() to overwrite the stack back trace 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 back trace 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.getPropagatedExceptionClass() method. Furthermore, the association for the thread-local stack back trace buffer is changed from the Throwable that crossed its scope boundary to the ThrowBoundaryError.
- If the current contents of the captured stack back trace information is associated with the Class returned from this ThrowBoundaryError object's getPropagatedExceptionClass() method, then the implementation of the ThrowBoundaryError.getPropagatedExceptionClass() method copies the contents of the stack back trace buffer at the time of its invocation. Otherwise, ThrowBoundaryError.getPropagatedExceptionClass() returns a zero-element array.
- 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 Services.captureBackTrace() to overwrite the stack back trace associated within the current thread with the preallocated exception.

SCJ defines its own ThrowBoundaryError class to stress that it works differently than the one in RTSJ and to provide some additional methods. The ThrowBoundaryError exception behaves as if it 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.

11.5 Compatibility

11.5.1 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.

11.5.2 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 safety-critical systems, any libraries that the system uses must also be certifiable. Given the costs of the certification process, it is desirable to keep the size of any 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 from the official standard for particular applications. In addition, many safety-critical software systems are missing certain features, such as file systems and networks. Therefore, the standard needs to accommodate both systems that have 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 will not be 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 class libraries for safety-critical applications.

The remainder of this chapter summarizes the differences between the SCJ specification and JDK 1.6. Where differences exist, a brief discussion of the rationale is provided.

Class	Level	Completeness	Rationale
Foo	2	Full	4
Bar	1	Partial	5

12.1 Comparison of SCJ with JDK 1.6 java.io

Within the `java.io package`, the only definition provided by the SCJ specification is the Serializable interface. This interface is the same as JDK 1.6.

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.

12.2 Comparison of SCJ with JDK 1.6 java.lang package

`Appendable` interface: SCJ specification is the same as JDK 1.6.

`CharSequence` interface: SCJ specification is the same as JDK 1.6.

`Cloneable` interface: is omitted from SCJ specification. Though it is often desirable to make deep copies of certain objects when manipulating these objects within nested memory scopes, it has been determined that the Cloneable interface does not represent a reliable way to accomplish this.

`Comparable` interface: SCJ specification is the same as JDK 1.6.

`Iterable` interface present in JDK 1.6 is not included in SCJ to reduce size and complexity.

`Readable` interface present in JDK 1.6 is not included in the SCJ specification to reduce size and complexity.

`Runnable` interface: SCJ is the same as JDK 1.6.

`Thread.UncaughtExceptionHandler` interface: SCJ is the same as JDK 1.6.

`class Boolean`: SCJ is the same as JDK 1.6.

`class Byte`: SCJ is the same as JDK 1.6.

`class Character`: SCJ is the same as JDK 1.6 except that the SCJ specification version does not define the following fields:

```
$DIRECTIONALITY_ARABIC_NUMBER$,
$DIRECTIONALITY_BOUNDARY_NEUTRAL$,
$DIRECTIONALITY_COMMON_NUMBER_SEPARATOR$,
$DIRECTIONALITY_EUROPEAN_NUMBER$,
$DIRECTIONALITY_EUROPEAN_NUMBER_SEPARATOR$,
$DIRECTIONALITY_LEFT_TO_RIGHT$,
$DIRECTIONALITY_LEFT_TO_RIGHT_EMBEDDING$,
$DIRECTIONALITY_LEFT_TO_RIGHT_OVERRIDE$,
$DIRECTIONALITY_NONSPACING_MARK$,
$DIRECTIONALITY_OTHER_NEUTRALS$,
$DIRECTIONALITY_PARAGRAPH_SEPARATOR$,
$DIRECTIONALITY_POP_DIRECTIONAL_FORMAT$,
$DIRECTIONALITY_RIGHT_TO_LEFT$,
$DIRECTIONALITY_RIGHT_TO_LEFT_ARABIC$,
$DIRECTIONALITY_RIGHT_TO_LEFT_EMBEDDING$,
$DIRECTIONALITY_RIGHT_TO_LEFT_OVERRIDE$,
$DIRECTIONALITY_SEGMENT_SEPARATOR$,
$DIRECTIONALITY_UNDEFINED$,
$DIRECTIONALITY_WHITESPACE$,
$MAX_CODE_POINT$,
$MAX_HIGH_SURROGATE$,
$MAX_LOW_SURROGATE$,
$MAX_SURROGATE$,
$MIN_CODE_POINT$,
$MIN_HIGH_SURROGATE$,
$MIN_LOW_SURROGATE$,
$MIN_SUPPLEMENTARY_CODE_POINT$,
$MIN_SURROGATE$
```

Nor does it define the following methods:

```
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)`

The rationale for these various omissions is to reduce the size and complexity of the `Character` class. It has been determined that safety-critical code would generally not be involved with significant amounts of text processing.

The class `Character.Subset` is omitted from the SCJ specification. The rationale for this omission is to reduce the size and complexity of the `java.lang` package. It has been determined that safety-critical code would generally not be involved with significant amounts of text processing.

The class `Character.UnicodeBlock` is omitted from the SCJ. The rationale for this omission is to reduce the size and complexity of the `java.lang package`. It has been determined that safety-critical code would generally not be involved with significant amounts of text processing.

The class `Class`: the SCJ specification does not implement

`AnnotatedElement`, `GenericDeclaration`, or `Type`.

The SCJ specification omits 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().
```

The rationale for these various omissions is to reduce the size and complexity of the Class class. It has been decided that SCJ should severely restrict reflection.

Note that `Class` class does implement the following methods:

```
getEnumConstants(),
getSuperclass(),
isAnnotation(),
isArray(),
isAssignableFrom(Class),
isEnum(),
isInstance(Object),
isInterface(),
newInstance(),
toString().
```

The class `ClassLoader` is omitted from the SCJ. It has been determined that dynamic class loading should not be supported by SCJ in order to reduce system size and complexity.

The class `Compiler` is omitted from the SCJ safety-critical Java specification. It has been determined that compilation, if any, of SCJ applications would normally be done at build time rather than at execution time. Removing this class reduces the size and complexity of a conformant SCJ run-time environment.

The class `Double`: SCJ specification is the same as JDK 1.6.

The class `Enum`: SCJ specification is the same as JDK 1.6 except that SCJ specification does not have `final finalize()` or `valueof(Class<T> enumType, String name)` methods.

The class `Float`: SCJ specification is the same as JDK 1.6.

The class `InheritableThreadLocal` is omitted to reduce the size and complexity of the SCJ specification.

The class `Integer`: SCJ specification is the same as JDK 1.6.

The class `Long`: SCJ specification is the same as JDK 1.6.

The class `Math`: SCJ specification is the same as JDK 1.6.

The class `Number`: SCJ specification is the same as JDK 1.6.

The class `Object`: SCJ specification considers the `finalize()` method to be not `@SCJAllowed`. This means safety-critical programmers should not override this method.

The following methods:

`notify()`,
`notifyAll()`,
`wait()`,
`wait(long timeout)`,
and `wait(long timeout, int nanos)`

are only `@SCJAllowed` at Level 2. It has been decided that the use of these services should be limited in order to enable a simpler run-time environment and easier analysis of real-time schedulability in Level 0 and Level 1. The `clone()` method is not `@SCJAllowed` as its default shallow-copy behavior is not compatible with typical scoped memory usage patterns.

The class `Package` is omitted from the SCJ specification. Reflection has been severely limited in the SCJ specification in order to reduce size and complexity.

The class `Process` is omitted from the SCJ specification. The services offered by this class will normally not be available within safety-certifiable operating environments.

The class `ProcessBuilder` is omitted from the SCJ specification. The services offered by this class will normally not be available within safety-certifiable operating environments.

The class `Runtime` is omitted from the SCJ specification. The services offered by this class will normally not be available within safety-certifiable operating environments and/or are not relevant in the absence of garbage collection and finalization.

The class `RuntimePermission` is omitted from the SCJ specification. This class is not relevant because SCJ does not support on-the-fly security management. In general, it is expected that safety-critical programs will assure security using static rather than dynamic techniques.

The class `SecurityManager` is omitted from the SCJ specification. This class is not relevant because SCJ does not support on-the-fly security management. In general, it is expected that safety-critical programs will assure security using static rather than dynamic techniques.

The class `Short`: SCJ specification is the same as JDK 1.6.

The class `StackTraceElement`: SCJ specification is the same as JDK 1.6.

The class `StrictMath`: SCJ specification is the same as JDK 1.6.

The class `String`: SCJ specification omits these constructors:

```
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.
```

SCJ specification also omits the methods:

```
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 SCJ specification is also omits `$CASE_INSENSITIVE_ORDER$` field.

The rationale for these various omissions is to reduce the size and complexity of the String class. It has been determined that safety-critical programs will not do extensive text processing.

The class `StringBuffer` is omitted from the SCJ specification. SCJ assumes a JDK 1.6 Java compiler, which generates uses of `StringBuilder` instead of `StringBuffer`.

The class `StringBuilder`: The SCJ specification omits the following methods:

```
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()
```

The rationale for these various omissions is to reduce the size and complexity of the `StringBuilder` class 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 class `System`: SCJ specification omits the following fields:

`err`,
`in`,
or `out`

Also, SCJ specification omits the following methods:

```
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)
```

The rationale for these various omissions is to reduce the size and complexity of the `System` class. Note that SCJ does not support garbage collection, security management, or file I/O.

The class `Thread`: SCJ specification does not implement the `Thread.State internal` class. The `Thread.UncaughtExceptionHandler` interface is the same as JDK 1.6. The SCJ specification does not implement the

`$MAX_PRIORITY$`,
`$MIN_PRIORITY$`,
or `$NORM_PRIORITY$` fields.

None of the constructors are `@SCJAllowed`. Only two constructors (`Thread()`, and `Thread(String)`) are available as `@SCJProtected`. The SCJ specification omits the following methods:

```
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()`

The rationale for these various omissions is to reduce the size and complexity of the Thread class. Note that SCJ does not allow instantiation of Threads because it only allows execution of `NoHeapRealtimeThreads`.

The class `ThreadGroup` is omitted from the SCJ specification in order to reduce the size and complexity of the SCJ specification.

The class `ThreadLocal` is omitted from the SCJ specification in order to reduce the size and complexity of the SCJ specification.

The class `Throwable`: The SCJ specification omits the following methods:

`fillInStackTrace()`, `getLocalizedMessage()`, `initCause(Throwable)`, `printStackTrace()`,`printStackTrace(PrintStream)`,`printStackTrace(PrintWriter)`, `setStackTrace()`,`toString()` methods. `Throwable` inherits a simple `toString()` method from Object.

The rationale for these various omissions is to reduce the size and complexity of the `Throwable` class and subclasses.

The class `ArithmeticException`: SCJ specification is the same as JDK 1.6.

The class `ArrayIndexOutOfBoundsException`: SCJ specification is the same as JDK 1.6.

The class `ArrayStoreException`: SCJ specification is the same as JDK 1.6.

The class `ClassCastException`: SCJ specification is the same as JDK 1.6.

The class `ClassNotFoundException`: SCJ specification is the same as JDK 1.6.

The class `CloneNotSupportedException`: SCJ specification is the same as JDK 1.6.

The class `EnumConstantNotPresentException` is omitted from the SCJ specification.

The class `Exception`: SCJ specification is the same as JDK 1.6.

The class `IllegalAccessException` is omitted from the SCJ specification. This exception is not needed in SCJ because SCJ does not support reflection.

The class `IllegalArgumentException`: SCJ specification is the same as JDK 1.6.

The class `IllegalMonitorStateException`: SCJ specification is same as JDK 1.6 and this is only allowed in Level 2.

The class `IllegalStateException`: SCJ specification is the same as JDK 1.6.

The class `IndexOutOfBoundsException`: SCJ specification is the same as JDK 1.6.

The class `InstantiationException`: SCJ specification is the same as JDK 1.6.

The class `InterruptedException`: SCJ specification is the same as JDK 1.6.

The class `NegativeArraySizeException`: SCJ specification is the same as JDK 1.6.

The class `NoSuchFieldException` is omitted from the SCJ specification. This exception is not relevant because SCJ does not support dynamic class loading.

The class `NoSuchMethodException` is omitted from the SCJ specification. This exception is not relevant because SCJ does not support dynamic class loading.

The class `NullPointerException`: SCJ specification is the same as JDK 1.6.

The class `NumberFormatException`: SCJ specification is the same as JDK 1.6.

The class `RuntimeException`: SCJ specification is the same as JDK 1.6.

The class `SecurityException` is omitted from the SCJ specification. This exception is not relevant because SCJ does not support dynamic security management.

The class `StringIndexOutOfBoundsException`: SCJ specification is the same as JDK 1.6.

The class `TypeNotPresentException` is omitted from the SCJ specification. This exception is not relevant because SCJ does not support reflection.

The class `UnsupportedOperationException`: SCJ specification is the same as JDK 1.6.

The class `AbstractMethodError` is omitted from the SCJ specification. This exception is not relevant because it can only arise during dynamic class loading.

The class `AssertionError`: SCJ specification is the same as JDK 1.6.

The class `ClassCircularityError` is omitted from the SCJ specification. This exception is not relevant because it can only arise during dynamic class loading.

The class `ClassFormatError` is omitted from the SCJ specification. This exception is not relevant because it can only arise during dynamic class loading.

The class `Error`: SCJ specification is the same as JDK 1.6.

The class `ExceptionInInitializerError` is omitted from the SCJ specification.

The class `IllegalAccessError` is omitted from the SCJ specification. This exception is not relevant because it can only arise as a result of dynamic class loading.

The class `IncompatibleClassChangeError`: SCJ specification is the same as JDK 1.6. This may be thrown by an invoke interface operation because the Java byte-code verifier does not enforce that interface variables actually hold instance of the interface type.

This exception is not relevant because it can only arise as a result of dynamic class loading.

The class `InstantiationError` is omitted from the SCJ specification. This exception is not relevant because it can only arise as a result of dynamic class loading.

The class `InternalError`: SCJ specification is the same as JDK 1.6.

The class `LinkageError` is omitted from the SCJ specification. This exception is not relevant because it can only arise as a result of dynamic class loading.

The class `NoClassDefFoundError` is omitted from the SCJ specification. This exception is not relevant because it can only arise as a result of dynamic class loading.

The class `NoSuchFieldError` is omitted from the SCJ specification. This exception is not relevant because it can only arise as a result of dynamic class loading.

The class `NoSuchMethodError` is omitted from the SCJ specification. This exception is not relevant because it can only arise as a result of dynamic class loading.

The class `OutOfMemoryError`: SCJ specification is the same as JDK 1.6.

The class `StackOverflowError`: SCJ specification is the same as JDK 1.6.

The class `ThreadDeath` is omitted from the SCJ specification. This exception is not relevant because SCJ does not support the `Thread.stop()` method.

The class `UnknownError` is omitted from the SCJ safety-critical Java specification.

The class `UnsatisfiedLinkError`: SCJ is the same as JDK 1.6. This may be thrown upon invocation of a native method for which there is no known implementation.

The class `UnsupportedClassVersionError` is omitted from the SCJ specification. This exception is not relevant because it can only arise as a result of dynamic class loading.

The class `VerifyError` is omitted from the SCJ specification. This exception is not relevant because it can only arise as a result of dynamic class loading.

The class `VirtualMachineError`: SCJ specification is the same as JDK 1.6.

The class `Deprecated`: SCJ specification is the same as JDK 1.6.

The class `Override`: SCJ specification is the same as JDK 1.6.

The class `SuppressWarnings`: SCJ specification is the same as JDK 1.6.

12.3 Comparison of SCJ API with JDK 1.6 java.lang.annotation

The interface `Annotation`: SCJ specification is the same as JDK 1.6.

The enum `ElementType`: SCJ defines the same constants as JDK 1.6. (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 1.6. (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 1.6.

The class `Inherited`: SCJ specification is the same as JDK 1.6.

The class `Retention`: SCJ specification is the same as JDK 1.6.

The class `Target`: SCJ specification is the same as JDK 1.6.

12.4 Comparison of SCJ Safety-Critical Java API with JDK 1.6 java.util

Within the `java.util package`, the only definition provided by the SCJ specification is the `Iterator` interface. This interface is the same as JDK 1.6.

Appendix A

Javadoc Description of Package java.io

Package Contents *Page*

A.1 Interfaces

A.1.1 INTERFACE **Closeable**

@SCJAllowed
public interface Closeable

Unless specified to the contrary, see JDK 1.6 documentation.

Methods

@SCJAllowed
public void close( )

A.1.2 INTERFACE **DataInput**

@SCJAllowed
public interface DataInput

Methods

@SCJAllowed
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.

@SCJAllowed
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.

@SCJAllowed
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.

@SCJAllowed
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.

@SCJAllowed
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.

@SCJAllowed
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.

@SCJAllowed
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.

@SCJAllowed
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) | &#32;((c & 0xff) << 8) | (d & 0xff)) This method is suitable for reading bytes written by the writeInt method of interface DataOutput.

@SCJAllowed
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.

@SCJAllowed
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.

@SCJAllowed
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.

@SCJAllowed
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.

@SCJAllowed
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.

@SCJAllowed
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.

A.1.3 INTERFACE **DataOutput**

@SCJAllowed
public interface DataOutput

Methods

@SCJAllowed
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.

@SCJAllowed
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.

@SCJAllowed
public void writeBoolean(**boolean** v)
  throws java.io.IOException

Writes a boolean to the underlying output stream as a 1-byte value.

@SCJAllowed
public void writeByte(**int** v)
  throws java.io.IOException

Writes out a byte to the underlying output stream as a 1-byte value.

@SCJAllowed
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.

@SCJAllowed
public void writeChars(String s)
  throws java.io.IOException

Writes a string to the underlying output stream as a sequence of characters.

@SCJAllowed
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.

@SCJAllowed
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.

@SCJAllowed
public void writeInt(**int** v)
  throws java.io.IOException

Writes an int to the underlying output stream as four bytes, high byte first.

@SCJAllowed
public void writeLong(**long** v)
  throws java.io.IOException

Writes a long to the underlying output stream as eight bytes, high byte first.

@SCJAllowed
public void writeShort(**int** v)
  throws java.io.IOException

Writes a short to the underlying output stream as two bytes, high byte first.

@SCJAllowed
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.

A.1.4 INTERFACE **Flushable**

@SCJAllowed
public interface Flushable

Unless specified to the contrary, see JDK 1.6 documentation.

Methods

@SCJAllowed
public void flush( )

A.1.5 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.2 Classes

A.2.1 CLASS DataInputStream

@SCJAllowed
public class DataInputStream
  implements java.io.DataInput
  extends java.io.InputStream

Fields

@SCJAllowed
protected java.io.InputStream in

The input stream.

Constructors

@SCJAllowed
public DataInputStream(InputStream in)

Creates a DataInputStream and saves its argument, the input stream in, for later use.

Methods

@SCJAllowed
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.

@SCJAllowed
public void close( )

Closes this input stream and releases any system resources associated with the stream. This method simply performs in.close().

@SCJAllowed
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
public boolean markSupported( )

Tests if this input stream supports the mark and reset methods. This method simply performs in.markSupported().

@SCJAllowed
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.

@SCJAllowed
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.

@SCJAllowed
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
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.

@SCJAllowed
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.

@SCJAllowed
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.

@SCJAllowed
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.

@SCJAllowed
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.

@SCJAllowed
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.

@SCJAllowed
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.

@SCJAllowed
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.

@SCJAllowed
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.

@SCJAllowed
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.

@SCJAllowed
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.

@SCJAllowed
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

@SCJAllowed
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.

@SCJAllowed
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.

@SCJAllowed
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.

@SCJAllowed
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).

@SCJAllowed
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.

A.2.2 CLASS **DataOutputStream**

@SCJAllowed
public class DataOutputStream
  implements java.io.DataOutput
  extends java.io.OutputStream

Fields

@SCJAllowed
protected java.io.OutputStream out

Constructors

@SCJAllowed
public DataOutputStream(OutputStream out)

Methods

@SCJAllowed
public void close( )

Closes this output stream and releases any system resources associated with the stream.

@SCJAllowed
public void flush( )

Flushes this data output stream.

@SCJAllowed
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.

@SCJAllowed
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.

@SCJAllowed
public void writeBoolean(**boolean** v)
  throws java.io.IOException

Writes a boolean to the underlying output stream as a 1-byte value.

@SCJAllowed
public void writeByte(**int** v)
  throws java.io.IOException

Writes out a byte to the underlying output stream as a 1-byte value.

@SCJAllowed
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.

@SCJAllowed
public void writeChars(String s)
  throws java.io.IOException

Writes a string to the underlying output stream as a sequence of characters.

@SCJAllowed
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.

@SCJAllowed
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.

@SCJAllowed
public void writeInt(**int** v)
  throws java.io.IOException

Writes an int to the underlying output stream as four bytes, high byte first.

@SCJAllowed
public void writeLong(**long** v)
  throws java.io.IOException

Writes a long to the underlying output stream as eight bytes, high byte first.

```
@SCJAllowed
public void writeShort(int v)
  throws java.io.IOException
```

Writes a short to the underlying output stream as two bytes, high byte first.

```
@SCJAllowed
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.

A.2.3 CLASS **EOFException**

```
@SCJAllowed
public class EOFException
  implements java.io.Serializable
  extends java.io.IOException
```

Constructors

```
@SCJAllowed
@SCJRestricted({javax.safetycritical.annotate.Phase.ALL},
   maySelfSuspend = false,
   mayAllocate = true)
public EOFException( )
```

Shall not copy "this" to any instance or static field.

Invokes System.captureStackBacktrace(this) to save the back trace associated with the current thread.

```
@SCJAllowed
@SCJRestricted({javax.safetycritical.annotate.Phase.ALL},
   maySelfSuspend = false,
   mayAllocate = true)
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 Illegal-AssignmentError will be thrown.

Invokes System.captureStackBacktrace(this) to save the back trace associated with the current thread.

Memory behavior: This constructor requires that the "msg" argument reside in a scope that encloses the scope of the "this" argument.

A.2.4 CLASS **FilterOutputStream**

@SCJAllowed
public class FilterOutputStream **extends** java.io.OutputStream

Unless specified to the contrary, see JDK 1.6 documentation.

Constructors

```
@SCJAllowed
@SCJRestricted({javax.safetycritical.annotate.Phase.ALL},
    maySelfSuspend = false,
    mayAllocate = true)
public FilterOutputStream(OutputStream out)
```

Memory behavior: This constructor requires that the "out" argument reside in a scope that encloses the scope of the "this" argument.

Methods

```
@SCJAllowed
public void close( )
```

```
@SCJAllowed
public void flush( )
```

```
@SCJAllowed
public void write(byte [] b)
  throws java.io.IOException
```

```
@SCJAllowed
public void write(byte [] b, int off, int len)
  throws java.io.IOException
```

@SCJAllowed
public void write(**int** b)
  throws java.io.IOException

A.2.5 CLASS **IOException**

@SCJAllowed
public class IOException
  implements java.io.Serializable
  extends java.lang.Exception

Constructors

@SCJAllowed
@SCJRestricted({javax.safetycritical.annotate.Phase.ALL},
   maySelfSuspend = false,
   mayAllocate = true)
public IOException( )

Shall not copy "this" to any instance or static field.

Invokes System.captureStackBacktrace(this) to save the back trace associated with the current thread.

@SCJAllowed
@SCJRestricted({javax.safetycritical.annotate.Phase.ALL},
   maySelfSuspend = false,
   mayAllocate = true)
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 Illegal-AssignmentError will be thrown.

Invokes System.captureStackBacktrace(this) to save the back trace associated with the current thread.

Memory behavior: This constructor requires that the "msg" argument reside in a scope that encloses the scope of the "this" argument.

A.2.6 CLASS **InputStream**

@SCJAllowed
public abstract class InputStream
  implements java.io.Closeable
  extends java.lang.Object

Unless specified to the contrary, see JDK 1.6 documentation.

Constructors

@SCJAllowed
@SCJRestricted({javax.safetycritical.annotate.Phase.ALL},
  maySelfSuspend = false,
  mayAllocate = true)
public InputStream( )

Methods

@SCJAllowed
public int available( )

@SCJAllowed
public void close( )

@SCJAllowed
public void mark(**int** readlimit)

@SCJAllowed
public boolean markSupported( )

@SCJAllowed
public int read(**byte** [] b)
  throws java.io.IOException

@SCJAllowed
public int read(**byte** [] b, **int** off, **int** len)
  throws java.io.IOException

@SCJAllowed
public abstract int read( )

@SCJAllowed
public void reset( )

@SCJAllowed
public long skip(**long** n)
  throws java.io.IOException

A.2.7 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 1.6 documentation.

Constructors

@SCJAllowed
@SCJRestricted({javax.safetycritical.annotate.Phase.ALL},
  maySelfSuspend = false,
  mayAllocate = true)
public OutputStream( )

Methods

@SCJAllowed
public void close( )

@SCJAllowed
public void flush( )

@SCJAllowed
public void write(**byte** [] b)
  throws java.io.IOException

@SCJAllowed
public void write(**byte** [] b, **int** off, **int** len)
  throws java.io.IOException

@SCJAllowed
public abstract void write(**int** b)
  throws java.io.IOException

A.2.8 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
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
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
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
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
public void print(**int** i)

Print an integer. The string produced by {@link java.lang.String#valueOf(int)} is translated into bytes according to the platform's default character encoding, and these bytes are written in exactly the manner of the **write(int)** method.

i — The int to be printed.

See Also: java.lang.Integer.toString(int)

@SCJAllowed
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(int)** method.

s — The array of chars to be printed.

Throws NullPointerException If s is null

@SCJAllowed
public void print(Object obj)

Print an object. The string produced by the {@link java.lang.String#valueOf(Object)} 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(int)** method.

obj — The Object to be printed.

See Also: java.lang.Object.toString()

@SCJAllowed
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(int)** method.

s — The String to be printed.

@SCJAllowed
public void print(**long** l)

Print a long integer. The string produced by {@link java.lang.String#valueOf(long)} is translated into bytes according to the platform's default character encoding, and these bytes are written in exactly the manner of the **write(int)** method.

l — The long to be printed.

See Also: java.lang.Long.toString(long)

@SCJAllowed
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(int)** method.

c — The char to be printed.

@SCJAllowed
public void print(**boolean** b)

Print a boolean value. The string produced by {@link java.lang.String#valueOf(boolean)} is translated into bytes according to the platform's default character encoding, and these bytes are written in exactly the manner of the **write(int)** method.

b — The boolean to be printed.

@SCJAllowed
public void println(**boolean** x)

Print a boolean and then terminate the line. This method behaves as though it invokes **print(boolean)** and then **println()** .

x — The boolean to be printed.

@SCJAllowed
public void println(**char** x)

Print a character and then terminate the line. This method behaves as though it invokes **print(char)** and then **println()** .

x — The char to be printed.

@SCJAllowed
public void println(**int** x)

Print an integer and then terminate the line. This method behaves as though it invokes **print(int)** and then **println()** .

x — The int to be printed.

@SCJAllowed
public void println(**char** [] x)

Print an array of characters and then terminate the line. This method behaves as though it invokes **print(char[])** and then **println()** .

x — an array of chars to print.

@SCJAllowed
public void println(String x)

Print a String and then terminate the line. This method behaves as though it invokes **print(String)** and then **println()** .

x — The String to be printed.

@SCJAllowed
public void println(Object x)

Print an Object and then terminate the line. This method behaves as though it invokes **print(Object)** and then **println()** .

x — The Object to be printed.

@SCJAllowed
public void println(**long** x)

Print a long and then terminate the line. This method behaves as though it invokes **print(long)** and then **println()** .

x — a The long to be printed.

@SCJAllowed
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
protected void setError( )

Set the error state of the stream to true.

Since
JDK1.1

@SCJAllowed
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(char) or println(char) methods.

buf — A byte array.

off — Offset from which to start taking bytes.

len — Number of bytes to write.

@SCJAllowed
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(char) or println(char) methods.

b — The byte to be written.

See Also: java.io.PrintStream.print(char), java.io.PrintStream.println(char)

A.2.9 CLASS UTFDataFormatException

@SCJAllowed
public class UTFDataFormatException
  implements java.io.Serializable
  extends java.io.IOException

Constructors

@SCJAllowed
@SCJRestricted({javax.safetycritical.annotate.Phase.ALL},
   maySelfSuspend = false,
   mayAllocate = true)
public UTFDataFormatException( )

Shall not copy "this" to any instance or static field.

Invokes System.captureStackBacktrace(this) to save the back trace associated with the current thread.

@SCJAllowed
@SCJRestricted({javax.safetycritical.annotate.Phase.ALL},
   maySelfSuspend = false,
   mayAllocate = true)
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 Illegal-AssignmentError will be thrown.

Invokes System.captureStackBacktrace(this) to save the back trace associated with the current thread.

Memory behavior: This constructor requires that the "msg" argument reside in a scope that encloses the scope of the "this" argument.

Appendix B

Javadoc Description of Package java.lang

Package Contents *Page*

B.1 Interfaces

B.1.1 INTERFACE Appendable

@SCJAllowed
public interface Appendable

Methods

@SCJAllowed
public java.lang.Appendable append(CharSequence csq)

Memory behavior: This constructor may allocate objects within the same MemoryArea that holds the implicit this argument.

@SCJAllowed
public java.lang.Appendable append(CharSequence csq, **int** start, **int** end)

Memory behavior: This constructor may allocate objects within the same MemoryArea that holds the implicit this argument.

@SCJAllowed
public java.lang.Appendable append(**char** c)

Memory behavior: This constructor may allocate objects within the same MemoryArea that holds the implicit this argument.

B.1.2 INTERFACE CharSequence

@SCJAllowed
public interface CharSequence

Methods

@SCJAllowed @SCJRestricted(maySelfSuspend = false)
public char charAt(**int** index)

Implementations of this method must not allocate memory and must not allow "this" to escape the local variables.

@SCJAllowed @SCJRestricted(maySelfSuspend = false)
public int length( )

Implementations of this method must not allocate memory and must not allow "this" to escape the local variables.

@SCJAllowed @SCJRestricted(maySelfSuspend = false)
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.

Memory behavior: This constructor may allocate objects within the currently active MemoryArea.

@SCJAllowed @SCJRestricted(maySelfSuspend = false)
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.

Memory behavior: This constructor may allocate objects within the currently active MemoryArea.

B.1.3 INTERFACE **Cloneable**

```
@SCJAllowed
public interface Cloneable
```

Author
jjh

B.1.4 INTERFACE **Comparable**

```
@SCJAllowed
public interface Comparable<T>
```

Methods

```
@SCJAllowed
@SCJRestricted({javax.safetycritical.annotate.Phase.ALL},
   maySelfSuspend = false,
   mayAllocate = true)
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.

B.1.5 INTERFACE **Runnable**

```
@SCJAllowed
public interface Runnable
```

Methods

```
@SCJAllowed
public void run( )
```

The implementation of this method may, in general, perform allocations in immortal memory.

Memory behavior: This constructor may allocate objects within the currently active MemoryArea. This constructor may allocate objects within the ImmortalMemory MemoryArea. This constructor may allocate objects within the currently active mission's MissionMemory MemoryArea. This constructor may allocate objects within the same MemoryArea that holds the implicit this argument. This constructor may allocate a PrivateMemory area which consumes backing store memory associated with the current thread.

B.1.6 INTERFACE **Thread.UncaughtExceptionHandler**

@SCJAllowed(javax.safetycritical.annotate.Level.LEVEL_2)
public static interface Thread.UncaughtExceptionHandler

Methods

@SCJAllowed(javax.safetycritical.annotate.Level.LEVEL_2)
public void uncaughtException(Thread t, Throwable e)

Memory behavior: Allocates no memory. Does not allow implicit argument this, or explicit arguments t and e to escape local variables.

B.1.7 INTERFACE **UncaughtExceptionHandler**

@SCJAllowed
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
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.2 Classes

B.2.1 CLASS **ArithmeticException**

```
@SCJAllowed
public class ArithmeticException
  implements java.io.Serializable
  extends java.lang.RuntimeException
```

Constructors

```
@SCJAllowed
@SCJRestricted({javax.safetycritical.annotate.Phase.ALL},
   maySelfSuspend = false,
   mayAllocate = true)
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).

Memory behavior: This constructor may allocate objects within the currently active MemoryArea.

```
@SCJAllowed
@SCJRestricted({javax.safetycritical.annotate.Phase.ALL},
   maySelfSuspend = false,
   mayAllocate = true)
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 Illegal-AssignmentError will be thrown.

Allocates an application- and implementation-dependent amount of memory in the current scope (to represent stack backtrace).

Memory behavior: This constructor may allocate objects within the currently active MemoryArea.

This constructor requires that the "msg" argument reside in a scope that encloses the scope of the "this" argument.

B.2.2 CLASS **ArrayIndexOutOfBoundsException**

```
@SCJAllowed
public class ArrayIndexOutOfBoundsException
  implements java.io.Serializable
  extends java.lang.IndexOutOfBoundsException
```

Constructors

```
@SCJAllowed
@SCJRestricted({javax.safetycritical.annotate.Phase.ALL},
   maySelfSuspend = false,
   mayAllocate = true)
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).

Memory behavior: This constructor may allocate objects within the currently active MemoryArea.

```
@SCJAllowed
@SCJRestricted({javax.safetycritical.annotate.Phase.ALL},
   maySelfSuspend = false,
   mayAllocate = true)
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).

Memory behavior: This constructor may allocate objects within the currently active MemoryArea.

```
@SCJAllowed
@SCJRestricted({javax.safetycritical.annotate.Phase.ALL},
   maySelfSuspend = false,
   mayAllocate = true)
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).

Memory behavior: This constructor may allocate objects within the currently active MemoryArea.

This constructor requires that the "msg" argument reside in a scope that encloses the scope of the "this" argument.

B.2.3 CLASS **ArrayStoreException**

```
@SCJAllowed
public class ArrayStoreException
  implements java.io.Serializable
  extends java.lang.RuntimeException
```

Constructors

```
@SCJAllowed
@SCJRestricted({javax.safetycritical.annotate.Phase.ALL},
   maySelfSuspend = false,
   mayAllocate = true)
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).

Memory behavior: This constructor may allocate objects within the currently active MemoryArea.

```
@SCJAllowed
@SCJRestricted({javax.safetycritical.annotate.Phase.ALL},
   maySelfSuspend = false,
   mayAllocate = true)
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 Illegal-AssignmentError will be thrown.

Allocates an application- and implementation-dependent amount of memory in the current scope (to represent stack backtrace).

Memory behavior: This constructor may allocate objects within the currently active MemoryArea.

This constructor requires that the "msg" argument reside in a scope that encloses the scope of the "this" argument.

B.2.4 CLASS AssertionError

```
@SCJAllowed
public class AssertionError
  implements java.io.Serializable
  extends java.lang.Error
```

Constructors

```
@SCJAllowed
@SCJRestricted({javax.safetycritical.annotate.Phase.ALL},
   maySelfSuspend = false,
   mayAllocate = true)
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).

Memory behavior: This constructor may allocate objects within the currently active MemoryArea.

```
@SCJAllowed
@SCJRestricted({javax.safetycritical.annotate.Phase.ALL},
   maySelfSuspend = false,
   mayAllocate = true)
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).

Memory behavior: This constructor may allocate objects within the currently active MemoryArea.

```
@SCJAllowed
@SCJRestricted({javax.safetycritical.annotate.Phase.ALL},
   maySelfSuspend = false,
   mayAllocate = true)
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).

Memory behavior: This constructor may allocate objects within the currently active MemoryArea.

```
@SCJAllowed
@SCJRestricted({javax.safetycritical.annotate.Phase.ALL},
    maySelfSuspend = false,
    mayAllocate = true)
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).

Memory behavior: This constructor may allocate objects within the currently active MemoryArea.

```
@SCJAllowed
@SCJRestricted({javax.safetycritical.annotate.Phase.ALL},
    maySelfSuspend = false,
    mayAllocate = true)
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).

Memory behavior: This constructor may allocate objects within the currently active MemoryArea.

```
@SCJAllowed
@SCJRestricted({javax.safetycritical.annotate.Phase.ALL},
    maySelfSuspend = false,
    mayAllocate = true)
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).

Memory behavior: This constructor may allocate objects within the currently active MemoryArea.

```
@SCJAllowed
@SCJRestricted({javax.safetycritical.annotate.Phase.ALL},
   maySelfSuspend = false,
   mayAllocate = true)
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).

Memory behavior: This constructor may allocate objects within the currently active MemoryArea.

```
@SCJAllowed
@SCJRestricted({javax.safetycritical.annotate.Phase.ALL},
   maySelfSuspend = false,
   mayAllocate = true)
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).

Memory behavior: This constructor may allocate objects within the currently active MemoryArea.

This constructor requires that the "o" argument reside in a scope that encloses the scope of the "this" argument.

B.2.5 CLASS **Boolean**

@SCJAllowed
public class Boolean
  implements java.lang.Comparable, java.io.Serializable
  extends java.lang.Object

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
@SCJRestricted({javax.safetycritical.annotate.Phase.ALL},
  maySelfSuspend = false,
  mayAllocate = true)
public Boolean(**boolean** v)

Allocates no memory. Does not allow "this" to escape local variables.

@SCJAllowed
@SCJRestricted({javax.safetycritical.annotate.Phase.ALL},
  maySelfSuspend = false,
  mayAllocate = true)
public Boolean(String str)

Allocates no memory. Does not allow "this" or "str" argument to escape local variables.

Methods

@SCJAllowed
@SCJRestricted({javax.safetycritical.annotate.Phase.ALL},
  maySelfSuspend = false,
  mayAllocate = true)
public boolean booleanValue( )

Allocates no memory. Does not allow "this" to escape local variables.

```
@SCJAllowed
@SCJRestricted({javax.safetycritical.annotate.Phase.ALL},
   maySelfSuspend = false,
   mayAllocate = true)
```
public int compareTo(**Boolean** b)

Allocates no memory. Does not allow "this" or argument "b" to escape local variables.

```
@Override
@SCJAllowed
@SCJRestricted({javax.safetycritical.annotate.Phase.ALL},
   maySelfSuspend = false,
   mayAllocate = true)
```
public boolean equals(Object obj)

Allocates no memory. Does not allow "this" or argument "obj" to escape local variables.

```
@SCJAllowed
@SCJRestricted({javax.safetycritical.annotate.Phase.ALL},
   maySelfSuspend = false,
   mayAllocate = true)
```
public static boolean getBoolean(String str)

Allocates no memory. Does not allow argument "str" to escape local variables.

```
@Override
@SCJAllowed
@SCJRestricted({javax.safetycritical.annotate.Phase.ALL},
   maySelfSuspend = false,
   mayAllocate = true)
```
public int hashCode( )

Allocates no memory. Does not allow "this" to escape local variables.

```
@SCJAllowed
@SCJRestricted({javax.safetycritical.annotate.Phase.ALL},
   maySelfSuspend = false,
   mayAllocate = true)
```
public static boolean parseBoolean(String str)

Allocates no memory. Does not allow argument "str" to escape local variables.

```
@SCJAllowed
@SCJRestricted({javax.safetycritical.annotate.Phase.ALL},
   maySelfSuspend = false,
   mayAllocate = true)
```
public static java.lang.String toString(**boolean** value)

Allocates no memory. Returns a String literal which resides at the scope of the Classloader that is responsible for loading the Boolean class.

```
@Override
@SCJAllowed
@SCJRestricted({javax.safetycritical.annotate.Phase.ALL},
   maySelfSuspend = false,
   mayAllocate = true)
```
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()).

Memory behavior: This constructor may allocate objects within the currently active MemoryArea.

```
@SCJAllowed
@SCJRestricted({javax.safetycritical.annotate.Phase.ALL},
   maySelfSuspend = false,
   mayAllocate = true)
```
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
@SCJRestricted({javax.safetycritical.annotate.Phase.ALL},
   maySelfSuspend = false,
   mayAllocate = true)
```
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.2.6 CLASS **Byte**

@SCJAllowed
public class Byte
  implements java.lang.Comparable, java.io.Serializable
  extends java.lang.Number

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
@SCJRestricted({javax.safetycritical.annotate.Phase.ALL},
   maySelfSuspend = false,
   mayAllocate = true)
public Byte(**byte** val)

Allocates no memory. Does not allow "this" to escape local variables.

@SCJAllowed
@SCJRestricted({javax.safetycritical.annotate.Phase.ALL},
   maySelfSuspend = false,
   mayAllocate = true)
public Byte(String str)
  throws java.lang.NumberFormatException

Allocates no memory. Does not allow "this" or "str" argument to escape local variables.

Methods

```
@SCJAllowed
@SCJRestricted({javax.safetycritical.annotate.Phase.ALL},
   maySelfSuspend = false,
   mayAllocate = true)
public byte byteValue( )
```

Allocates no memory. Does not allow "this" to escape local variables.

```
@SCJAllowed
@SCJRestricted({javax.safetycritical.annotate.Phase.ALL},
   maySelfSuspend = false,
   mayAllocate = true)
public int compareTo(Byte other)
```

Allocates no memory. Does not allow "this" or "other" argument to escape local variables.

```
@SCJAllowed
@SCJRestricted({javax.safetycritical.annotate.Phase.ALL},
   maySelfSuspend = false,
   mayAllocate = true)
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
@SCJRestricted({javax.safetycritical.annotate.Phase.ALL},
   maySelfSuspend = false,
   mayAllocate = true)
public double doubleValue( )
```

Allocates no memory. Does not allow "this" to escape local variables.

```
@SCJAllowed
@SCJRestricted({javax.safetycritical.annotate.Phase.ALL},
   maySelfSuspend = false,
   mayAllocate = true)
public boolean equals(Object obj)
```

Allocates no memory. Does not allow "this" or "obj" argument to escape local variables.

```
@SCJAllowed
@SCJRestricted({javax.safetycritical.annotate.Phase.ALL},
   maySelfSuspend = false,
   mayAllocate = true)
public float floatValue( )
```

Allocates no memory. Does not allow "this" to escape local variables.

```
@SCJAllowed
@SCJRestricted({javax.safetycritical.annotate.Phase.ALL},
   maySelfSuspend = false,
   mayAllocate = true)
public int hashCode( )
```

Allocates no memory. Does not allow "this" to escape local variables.

```
@SCJAllowed
@SCJRestricted({javax.safetycritical.annotate.Phase.ALL},
   maySelfSuspend = false,
   mayAllocate = true)
public int intValue( )
```

Allocates no memory. Does not allow "this" to escape local variables.

```
@SCJAllowed
@SCJRestricted({javax.safetycritical.annotate.Phase.ALL},
   maySelfSuspend = false,
   mayAllocate = true)
public long longValue( )
```

Allocates no memory. Does not allow "this" to escape local variables.

```
@SCJAllowed
@SCJRestricted({javax.safetycritical.annotate.Phase.ALL},
   maySelfSuspend = false,
   mayAllocate = true)
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
@SCJRestricted({javax.safetycritical.annotate.Phase.ALL},
    maySelfSuspend = false,
    mayAllocate = true)
public static byte parseByte(String str)
  throws java.lang.NumberFormatException
```

Allocates no memory. Does not allow "str" argument to escape local variables.

```
@SCJAllowed
@SCJRestricted({javax.safetycritical.annotate.Phase.ALL},
    maySelfSuspend = false,
    mayAllocate = true)
public short shortValue( )
```

Allocates no memory. Does not allow "this" to escape local variables.

```
@SCJAllowed
@SCJRestricted({javax.safetycritical.annotate.Phase.ALL},
    maySelfSuspend = false,
    mayAllocate = true)
public static java.lang.String toString(byte v)
```

Allocates a String and associated internal "structure" (e.g. char[]) in caller's scope.

Memory behavior: This constructor may allocate objects within the currently active MemoryArea.

```
@SCJAllowed
@SCJRestricted({javax.safetycritical.annotate.Phase.ALL},
    maySelfSuspend = false,
    mayAllocate = true)
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()).

Memory behavior: This constructor may allocate objects within the currently active MemoryArea.

```
@SCJAllowed
@SCJRestricted({javax.safetycritical.annotate.Phase.ALL},
   maySelfSuspend = false,
   mayAllocate = true)
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.

Memory behavior: This constructor may allocate objects within the currently active MemoryArea.

```
@SCJAllowed
@SCJRestricted({javax.safetycritical.annotate.Phase.ALL},
   maySelfSuspend = false,
   mayAllocate = true)
public static java.lang.Byte valueOf(byte val)
```

Allocates one Byte object in the caller's scope.

Memory behavior: This constructor may allocate objects within the currently active MemoryArea.

```
@SCJAllowed
@SCJRestricted({javax.safetycritical.annotate.Phase.ALL},
   maySelfSuspend = false,
   mayAllocate = true)
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.

Memory behavior: This constructor may allocate objects within the currently active MemoryArea.

B.2.7 CLASS **Character**

@SCJAllowed
public final class Character
  implements java.lang.Comparable, java.io.Serializable
  extends java.lang.Object

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
@SCJRestricted({javax.safetycritical.annotate.Phase.ALL},
   maySelfSuspend = false,
   mayAllocate = true)
```
public Character(**char** v)

Allocates no memory. Does not allow "this" to escape local variables.

Methods

```
@SCJAllowed
@SCJRestricted({javax.safetycritical.annotate.Phase.ALL},
   maySelfSuspend = false,
   mayAllocate = true)
```
public char charValue( )

Allocates no memory. Does not allow "this" to escape local variables.

```
@SCJAllowed
@SCJRestricted({javax.safetycritical.annotate.Phase.ALL},
   maySelfSuspend = false,
   mayAllocate = true)
```
public int compareTo(Character another_character)

Allocates no memory. Does not allow "this" or "another_character" argument to escape local variables.

```
@SCJAllowed
@SCJRestricted({javax.safetycritical.annotate.Phase.ALL},
   maySelfSuspend = false,
   mayAllocate = true)
```
public static int digit(**char** ch, **int** radix)

Allocates no memory.

```
@SCJAllowed
@SCJRestricted({javax.safetycritical.annotate.Phase.ALL},
   maySelfSuspend = false,
   mayAllocate = true)
```
public boolean equals(Object obj)

Allocates no memory. Does not allow "this" or "obj" argument to escape local variables.

@SCJAllowed
@SCJRestricted({javax.safetycritical.annotate.Phase.ALL},
    maySelfSuspend = false,
    mayAllocate = true)
public static int getType(**char** ch)

Allocates no memory.

@SCJAllowed
@SCJRestricted({javax.safetycritical.annotate.Phase.ALL},
    maySelfSuspend = false,
    mayAllocate = true)
public int hashCode( )

@SCJAllowed
@SCJRestricted({javax.safetycritical.annotate.Phase.ALL},
    maySelfSuspend = false,
    mayAllocate = true)
public static boolean isLetter(**char** ch)

Allocates no memory.

@SCJAllowed
@SCJRestricted({javax.safetycritical.annotate.Phase.ALL},
    maySelfSuspend = false,
    mayAllocate = true)
public static boolean isLetterOrDigit(**char** ch)

Allocates no memory.

@SCJAllowed
@SCJRestricted({javax.safetycritical.annotate.Phase.ALL},
    maySelfSuspend = false,
    mayAllocate = true)
public static boolean isLowerCase(**char** ch)

Allocates no memory.

@SCJAllowed
@SCJRestricted({javax.safetycritical.annotate.Phase.ALL},
    maySelfSuspend = false,
    mayAllocate = true)
public static boolean isSpaceChar(**char** ch)

Allocates no memory.

```
@SCJAllowed
@SCJRestricted({javax.safetycritical.annotate.Phase.ALL},
    maySelfSuspend = false,
    mayAllocate = true)
```
public static boolean isUpperCase(**char** ch)

Allocates no memory.

```
@SCJAllowed
@SCJRestricted({javax.safetycritical.annotate.Phase.ALL},
    maySelfSuspend = false,
    mayAllocate = true)
```
public static boolean isWhitespace(**char** ch)

Allocates no memory.

```
@SCJAllowed
@SCJRestricted({javax.safetycritical.annotate.Phase.ALL},
    maySelfSuspend = false,
    mayAllocate = true)
```
public static char toLowerCase(**char** ch)

Allocates no memory.

```
@SCJAllowed
@SCJRestricted({javax.safetycritical.annotate.Phase.ALL},
    maySelfSuspend = false,
    mayAllocate = true)
```
public static java.lang.String toString(**char** c)

Allocates a String and associated internal "structure" (e.g. char[]) in caller's scope.

Memory behavior: This constructor may allocate objects within the currently active MemoryArea.

@SCJAllowed
@SCJRestricted({javax.safetycritical.annotate.Phase.ALL},
  maySelfSuspend = false,
  mayAllocate = true)
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()).

Memory behavior: This constructor may allocate objects within the currently active MemoryArea.

@SCJAllowed
@SCJRestricted({javax.safetycritical.annotate.Phase.ALL},
  maySelfSuspend = false,
  mayAllocate = true)
public static char toUpperCase(**char** ch)

Allocates no memory.

@SCJAllowed
@SCJRestricted({javax.safetycritical.annotate.Phase.ALL},
  maySelfSuspend = false,
  mayAllocate = true)
public static java.lang.Character valueOf(**char** c)

Allocates a Character object in caller's scope.

Memory behavior: This constructor may allocate objects within the currently active MemoryArea.

B.2.8 CLASS **Class**

@SCJAllowed
public final class Class<T>
  implements java.io.Serializable
  extends java.lang.Object

Methods

@SCJAllowed
@SCJRestricted({javax.safetycritical.annotate.Phase.ALL},
    maySelfSuspend = false,
    mayAllocate = true)
public boolean desiredAssertionStatus( )

Phase.ALL,ocates no memory. Does not allow "this" to escape local variables.

@SCJAllowed
@SCJRestricted({javax.safetycritical.annotate.Phase.ALL},
    maySelfSuspend = false,
    mayAllocate = true)
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
@SCJRestricted({javax.safetycritical.annotate.Phase.ALL},
    maySelfSuspend = false,
    mayAllocate = true)
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
@SCJRestricted({javax.safetycritical.annotate.Phase.ALL},
    maySelfSuspend = false,
    mayAllocate = true)
public 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.

Memory behavior: This constructor may allocate objects within the currently active MemoryArea.

This constructor requires that the "this.getClass().getClassLoader()" argument reside in a scope that encloses the scope of the "@result" argument.

```
@SCJAllowed
@SCJRestricted({javax.safetycritical.annotate.Phase.ALL},
   maySelfSuspend = false,
   mayAllocate = true)
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
@SCJRestricted({javax.safetycritical.annotate.Phase.ALL},
   maySelfSuspend = false,
   mayAllocate = true)
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
@SCJRestricted({javax.safetycritical.annotate.Phase.ALL},
   maySelfSuspend = false,
   mayAllocate = true)
public boolean isAnnotation( )
```

Allocates no memory. Does not allow "this" to escape local variables.

```
@SCJAllowed
@SCJRestricted({javax.safetycritical.annotate.Phase.ALL},
   maySelfSuspend = false,
   mayAllocate = true)
public boolean isArray( )
```

Allocates no memory. Does not allow "this" to escape local variables.

@SCJAllowed
@SCJRestricted({javax.safetycritical.annotate.Phase.ALL},
    maySelfSuspend = false,
    mayAllocate = true)
public boolean isAssignableFrom(**Class**<> c)

Allocates no memory. Does not allow "this" or argument "c" to escape local variables.

@SCJAllowed
@SCJRestricted({javax.safetycritical.annotate.Phase.ALL},
    maySelfSuspend = false,
    mayAllocate = true)
public boolean isEnum( )

Allocates no memory. Does not allow "this" to escape local variables.

@SCJAllowed
@SCJRestricted({javax.safetycritical.annotate.Phase.ALL},
    maySelfSuspend = false,
    mayAllocate = true)
public boolean isInstance(Object o)

Allocates no memory. Does not allow "this" to escape local variables.

@SCJAllowed
@SCJRestricted({javax.safetycritical.annotate.Phase.ALL},
    maySelfSuspend = false,
    mayAllocate = true)
public boolean isInterface( )

Allocates no memory. Does not allow "this" to escape local variables.

@SCJAllowed
@SCJRestricted({javax.safetycritical.annotate.Phase.ALL},
    maySelfSuspend = false,
    mayAllocate = true)
public boolean isPrimitive( )

Allocates no memory. Does not allow "this" to escape local variables.

```
@SCJAllowed
@SCJRestricted({javax.safetycritical.annotate.Phase.ALL},
   maySelfSuspend = false,
   mayAllocate = true)
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()).

Memory behavior: This constructor may allocate objects within the currently active MemoryArea.

B.2.9 CLASS **ClassCastException**

```
@SCJAllowed
public class ClassCastException
  implements java.io.Serializable
  extends java.lang.RuntimeException
```

Constructors

```
@SCJAllowed
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
@SCJRestricted({javax.safetycritical.annotate.Phase.ALL},
   maySelfSuspend = false,
   mayAllocate = true)
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).

Memory behavior: This constructor may allocate objects within the currently active MemoryArea.

This constructor requires that the "msg" argument reside in a scope that encloses the scope of the "this" argument.

B.2.10 CLASS ClassNotFoundException

```
@SCJAllowed
public class ClassNotFoundException
  implements java.io.Serializable
  extends java.lang.Exception
```

Constructors

```
@SCJAllowed
@SCJRestricted({javax.safetycritical.annotate.Phase.ALL},
   maySelfSuspend = false,
   mayAllocate = true)
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).

Memory behavior: This constructor may allocate objects within the currently active MemoryArea.

```
@SCJAllowed
@SCJRestricted({javax.safetycritical.annotate.Phase.ALL},
   maySelfSuspend = false,
   mayAllocate = true)
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 Illegal-AssignmentError will be thrown.

Allocates an application- and implementation-dependent amount of memory in the current scope (to represent stack backtrace).

Memory behavior: This constructor may allocate objects within the currently active MemoryArea.

This constructor requires that the "msg" argument reside in a scope that encloses the scope of the "this" argument.

B.2.11 CLASS CloneNotSupportedException

```
@SCJAllowed
public class CloneNotSupportedException
  implements java.io.Serializable
  extends java.lang.Exception
```

Constructors

```
@SCJAllowed
@SCJRestricted({javax.safetycritical.annotate.Phase.ALL},
    maySelfSuspend = false,
    mayAllocate = true)
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).

Memory behavior: This constructor may allocate objects within the currently active MemoryArea.

```
@SCJAllowed
@SCJRestricted({javax.safetycritical.annotate.Phase.ALL},
    maySelfSuspend = false,
    mayAllocate = true)
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).

Memory behavior: This constructor may allocate objects within the currently active MemoryArea.

This constructor requires that the "msg" argument reside in a scope that encloses the scope of the "this" argument.

B.2.12 Class **Double**

@SCJAllowed
public class Double
  implements java.lang.Comparable, java.io.Serializable
  extends java.lang.Number

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
@SCJRestricted({javax.safetycritical.annotate.Phase.ALL},
    maySelfSuspend = false,
    mayAllocate = true)
public Double(**double** val)

Allocates no memory. Does not allow "this" to escape local variables.

@SCJAllowed
@SCJRestricted({javax.safetycritical.annotate.Phase.ALL},
    maySelfSuspend = false,
    mayAllocate = true)
public Double(String str)
  throws java.lang.NumberFormatException

Allocates no memory. Does not allow "this" or "str" argument to escape local variables.

Methods

@SCJAllowed
@SCJRestricted({javax.safetycritical.annotate.Phase.ALL},
    maySelfSuspend = false,
    mayAllocate = true)
public byte byteValue( )

Allocates no memory. Does not allow "this" to escape local variables.

@SCJAllowed
@SCJRestricted({javax.safetycritical.annotate.Phase.ALL},
  maySelfSuspend = false,
  mayAllocate = true)
public static int compare(**double** value1, **double** value2)

Allocates no memory.

@SCJAllowed
@SCJRestricted({javax.safetycritical.annotate.Phase.ALL},
  maySelfSuspend = false,
  mayAllocate = true)
public int compareTo(**Double** other)

Allocates no memory. Does not allow "this" or "other" argument to escape local variables.

@SCJAllowed
@SCJRestricted({javax.safetycritical.annotate.Phase.ALL},
  maySelfSuspend = false,
  mayAllocate = true)
public static long doubleToLongBits(**double** v)

Allocates no memory.

@SCJAllowed
@SCJRestricted({javax.safetycritical.annotate.Phase.ALL},
  maySelfSuspend = false,
  mayAllocate = true)
public static native long doubleToRawLongBits(**double** val)

Allocates no memory.

@SCJAllowed
@SCJRestricted({javax.safetycritical.annotate.Phase.ALL},
  maySelfSuspend = false,
  mayAllocate = true)
public double doubleValue( )

Allocates no memory. Does not allow "this" to escape local variables.

```
@SCJAllowed
@SCJRestricted({javax.safetycritical.annotate.Phase.ALL},
   maySelfSuspend = false,
   mayAllocate = true)
```
public boolean equals(Object obj)

Allocates no memory. Does not allow "this" or "obj" argument to escape local variables.

```
@SCJAllowed
@SCJRestricted({javax.safetycritical.annotate.Phase.ALL},
   maySelfSuspend = false,
   mayAllocate = true)
```
public float floatValue( )

Allocates no memory. Does not allow "this" to escape local variables.

```
@SCJAllowed
@SCJRestricted({javax.safetycritical.annotate.Phase.ALL},
   maySelfSuspend = false,
   mayAllocate = true)
```
public int hashCode( )

Allocates no memory. Does not allow "this" to escape local variables.

```
@SCJAllowed
@SCJRestricted({javax.safetycritical.annotate.Phase.ALL},
   maySelfSuspend = false,
   mayAllocate = true)
```
public int intValue( )

Allocates no memory. Does not allow "this" to escape local variables.

```
@SCJAllowed
@SCJRestricted({javax.safetycritical.annotate.Phase.ALL},
   maySelfSuspend = false,
   mayAllocate = true)
```
public static boolean isInfinite(**double** v)

Allocates no memory.

@SCJAllowed
@SCJRestricted({javax.safetycritical.annotate.Phase.ALL},
    maySelfSuspend = false,
    mayAllocate = true)
public boolean isInfinite( )

Allocates no memory. Does not allow "this" to escape local variables.

@SCJAllowed
@SCJRestricted({javax.safetycritical.annotate.Phase.ALL},
    maySelfSuspend = false,
    mayAllocate = true)
public static boolean isNaN(**double** v)

Allocates no memory.

@SCJAllowed
@SCJRestricted({javax.safetycritical.annotate.Phase.ALL},
    maySelfSuspend = false,
    mayAllocate = true)
public boolean isNaN( )

Allocates no memory. Does not allow "this" to escape local variables.

@SCJAllowed
@SCJRestricted({javax.safetycritical.annotate.Phase.ALL},
    maySelfSuspend = false,
    mayAllocate = true)
public static double longBitsToDouble(**long** v)

Allocates no memory.

@SCJAllowed
@SCJRestricted({javax.safetycritical.annotate.Phase.ALL},
    maySelfSuspend = false,
    mayAllocate = true)
public long longValue( )

Allocates no memory. Does not allow "this" to escape local variables.

```
@SCJAllowed
@SCJRestricted({javax.safetycritical.annotate.Phase.ALL},
    maySelfSuspend = false,
    mayAllocate = true)
public static double parseDouble(String s)
  throws java.lang.NumberFormatException
```

Does not allow "this" to escape local variables.

```
@SCJAllowed
@SCJRestricted({javax.safetycritical.annotate.Phase.ALL},
    maySelfSuspend = false,
    mayAllocate = true)
public short shortValue( )
```

Allocates no memory. Does not allow "this" to escape local variables.

```
@SCJAllowed
@SCJRestricted({javax.safetycritical.annotate.Phase.ALL},
    maySelfSuspend = false,
    mayAllocate = true)
public static java.lang.String toString(double v)
```

Allocates a String and associated internal "structure" (e.g. char[]) in caller's scope.

Memory behavior: This constructor may allocate objects within the currently active MemoryArea.

```
@SCJAllowed
@SCJRestricted({javax.safetycritical.annotate.Phase.ALL},
    maySelfSuspend = false,
    mayAllocate = true)
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()).

Memory behavior: This constructor may allocate objects within the currently active MemoryArea.

```
@SCJAllowed
@SCJRestricted({javax.safetycritical.annotate.Phase.ALL},
   maySelfSuspend = false,
   mayAllocate = true)
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.

Memory behavior: This constructor may allocate objects within the currently active MemoryArea.

```
@SCJAllowed
@SCJRestricted({javax.safetycritical.annotate.Phase.ALL},
   maySelfSuspend = false,
   mayAllocate = true)
public static java.lang.Double valueOf(double val)
```

Allocates a Double in caller's scope.

Memory behavior: This constructor may allocate objects within the currently active MemoryArea.

B.2.13 CLASS **Enum**

```
@SCJAllowed
public abstract class Enum<E extends Enum>
  implements java.lang.Comparable, java.io.Serializable
  extends java.lang.Object
```

Constructors

```
@SCJAllowed
@SCJRestricted({javax.safetycritical.annotate.Phase.ALL},
    maySelfSuspend = false,
    mayAllocate = true)
```
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".

Memory behavior: This constructor requires that the "name" argument reside in a scope that encloses the scope of the "this" argument.

Methods

```
@SCJRestricted({javax.safetycritical.annotate.Phase.ALL},
    maySelfSuspend = false,
    mayAllocate = true)
@SCJAllowed(javax.safetycritical.annotate.Level.HIDDEN)
```
protected final java.lang.Object clone( )

> Does not allow "this" to escape local variables.

Memory behavior: This constructor may allocate objects within the currently active MemoryArea.

```
@SCJAllowed
@SCJRestricted({javax.safetycritical.annotate.Phase.ALL},
    maySelfSuspend = false,
    mayAllocate = false)
```
public final int compareTo(E o)

> Allocates no memory. Does not allow "this" or "o" argument to escape local variables.

```
@SCJAllowed
@SCJRestricted({javax.safetycritical.annotate.Phase.ALL},
   maySelfSuspend = false,
   mayAllocate = false)
public final boolean equals(Object o)
```

Allocates no memory. Does not allow "this" or "o" argument to escape local variables.

```
@SCJAllowed
@SCJRestricted({javax.safetycritical.annotate.Phase.ALL},
   maySelfSuspend = false,
   mayAllocate = false)
public final java.lang.Class<E> getDeclaringClass( )
```

Allocates no memory. Returns a reference to a previously allocated Class, which resides in its ClassLoader scope.

```
@SCJAllowed
@SCJRestricted({javax.safetycritical.annotate.Phase.ALL},
   maySelfSuspend = false,
   mayAllocate = false)
public final int hashCode( )
```

Allocates no memory. Does not allow "this" to escape local variables.

```
@SCJAllowed
@SCJRestricted({javax.safetycritical.annotate.Phase.ALL},
   maySelfSuspend = false,
   mayAllocate = false)
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.

```
@SCJAllowed
@SCJRestricted({javax.safetycritical.annotate.Phase.ALL},
   maySelfSuspend = false,
   mayAllocate = false)
public final int ordinal( )
```

Allocates no memory. Does not allow "this" to escape local variables.

```
@SCJAllowed
@SCJRestricted({javax.safetycritical.annotate.Phase.ALL},
    maySelfSuspend = false,
    mayAllocate = true)
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()).

Memory behavior: This constructor may allocate objects within the currently active MemoryArea.

B.2.14 CLASS **Error**

```
@SCJAllowed
public class Error
  implements java.io.Serializable
  extends java.lang.Throwable
```

Constructors

```
@SCJAllowed @SCJRestricted(maySelfSuspend = false)
public Error( )
```

Constructs an Error object for an SCJ system with a null detail message. Invokes System.captureStackBacktrace(this) to save the back trace associated with the current thread.

Allocates an application- and implementation-defined amount of memory in the current scope (to represent stack backtrace).

This constructor shall not copy this to any instance or static field.

Memory behavior: This constructor may allocate objects within the currently active MemoryArea.

```
@SCJAllowed @SCJRestricted(maySelfSuspend = false)
public Error(String msg)
```

Constructs an Error object for an SCJ system with a detail message. Invokes System.captureStackBacktrace(this) to save the back trace associated with the current thread.

Allocates an application- and implementation-defined amount of memory in the current scope (to represent stack backtrace).

This constructor shall not copy this to any instance or static field.

msg — the detail message for this Error object.

Memory behavior: This constructor may allocate objects within the currently active MemoryArea.

This constructor requires that the "msg" argument reside in a scope that encloses the scope of the "this" argument.

@SCJAllowed @SCJRestricted(maySelfSuspend = false)
public Error(String msg, Throwable t)

Constructs an Error object for an SCJ system with a specified detail message and with a specified cause. Does not invoke System.captureStackBacktrace(this) to avoid overwriting the back trace associated with the cause.

Allocates an application- and implementation-defined amount of memory in the current scope (to represent stack backtrace).

This constructor shall not copy this to any instance or static field.

msg — the detail message for this Error object.

t — the exception that caused this error.

Memory behavior: This constructor may allocate objects within the currently active MemoryArea.

This constructor requires that the "msg" argument reside in a scope that encloses the scope of the "this" argument. This constructor requires that the "t" argument reside in a scope that encloses the scope of the "this" argument.

@SCJAllowed @SCJRestricted(maySelfSuspend = false)
public Error(Throwable t)

Constructs an Error object for an SCJ system with a null detail message and with a specified cause. Does not invoke System.captureStackBacktrace(this) to avoid overwriting the back trace associated with the cause.

Allocates an application- and implementation-defined amount of memory in the current scope (to represent stack backtrace).

This constructor shall not copy this to any instance or static field.

t — the exception that caused this error.

Memory behavior: This constructor may allocate objects within the currently active MemoryArea.

This constructor requires that the "t" argument reside in a scope that encloses the scope of the "this" argument.

B.2.15 CLASS **Exception**

@SCJAllowed
public class Exception
  implements java.io.Serializable
  extends java.lang.Throwable

Constructors

@SCJAllowed @SCJRestricted(maySelfSuspend = false)
public Exception( )

Constructs an Exception object for an SCJ system with a null detail message. Invokes System.captureStackBacktrace(this) to save the back trace associated with the current thread.

Allocates an application- and implementation-defined amount of memory in the current scope (to represent stack backtrace).

This constructor shall not copy this to any instance or static field.

Memory behavior: This constructor may allocate objects within the currently active MemoryArea.

@SCJAllowed @SCJRestricted(maySelfSuspend = false)
public Exception(String msg)

Constructs an Exception object for an SCJ system with a specified detail message. Invokes System.captureStackBacktrace(this) to save the back trace associated with the current thread.

This constructor shall not copy this to any instance or static field.

msg — the detail message for this Exception object.

Memory behavior: This constructor may allocate objects within the currently active MemoryArea.

This constructor requires that the "msg" argument reside in a scope that encloses the scope of the "this" argument.

@SCJAllowed @SCJRestricted(maySelfSuspend = false)
public Exception(Throwable cause)

Constructs an Exception object for an SCJ system with a null detail message and a specified cause. Does not invoke System.captureStackBacktrace(this) to avoid overwriting the back trace associated with the cause.

This constructor shall not copy this to any instance or static field.

cause — the cause of this exception.

Memory behavior: This constructor requires that the "cause" argument reside in a scope that encloses the scope of the "this" argument.

@SCJAllowed @SCJRestricted(maySelfSuspend = false)
public Exception(String msg, Throwable cause)

Constructs an Exception object for an SCJ system with a specified detail message and a specified cause. Does not invoke System.captureStackBacktrace(this) to avoid overwriting the back trace associated with the cause.

This constructor shall not copy this to any instance or static field.

msg — the detail message for this Exception object.

cause — the cause of this exception .

Memory behavior: This constructor requires that the "cause" argument reside in a scope that encloses the scope of the "this" argument. This constructor requires that the "msg" argument reside in a scope that encloses the scope of the "this" argument.

B.2.16 CLASS ExceptionInInitializerError

@SCJAllowed
public class ExceptionInInitializerError **extends** java.lang.Exception

Constructors

@SCJAllowed
public ExceptionInInitializerError( )

@SCJAllowed
public ExceptionInInitializerError(String msg)

@SCJAllowed
public ExceptionInInitializerError(Throwable cause)

@SCJAllowed
public ExceptionInInitializerError(String msg, Throwable cause)

B.2.17 CLASS Float

@SCJAllowed
public class Float
  implements java.lang.Comparable, java.io.Serializable
  extends java.lang.Number

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
@SCJRestricted({javax.safetycritical.annotate.Phase.ALL},
  maySelfSuspend = false,
  mayAllocate = true)
public Float(**float** val)

Allocates no memory. Does not allow "this" to escape local variables.

@SCJAllowed
@SCJRestricted({javax.safetycritical.annotate.Phase.ALL},
  maySelfSuspend = false,
  mayAllocate = true)
public Float(**double** val)

Allocates no memory. Does not allow "this" to escape local variables.

```
@SCJAllowed
@SCJRestricted({javax.safetycritical.annotate.Phase.ALL},
   maySelfSuspend = false,
   mayAllocate = true)
public Float(String str)
  throws java.lang.NumberFormatException
```

Allocates no memory. Does not allow "this" or "str" argument to escape local variables.

Methods

```
@SCJAllowed
@SCJRestricted({javax.safetycritical.annotate.Phase.ALL},
   maySelfSuspend = false,
   mayAllocate = true)
public byte byteValue( )
```

Allocates no memory. Does not allow "this" to escape local variables.

```
@SCJAllowed
@SCJRestricted({javax.safetycritical.annotate.Phase.ALL},
   maySelfSuspend = false,
   mayAllocate = true)
public static int compare(float value1, float value2)
```

Allocates no memory.

```
@SCJAllowed
@SCJRestricted({javax.safetycritical.annotate.Phase.ALL},
   maySelfSuspend = false,
   mayAllocate = true)
public int compareTo(Float other)
```

Allocates no memory. Does not allow "this" or "other" argument to escape local variables.

```
@SCJAllowed
@SCJRestricted({javax.safetycritical.annotate.Phase.ALL},
   maySelfSuspend = false,
   mayAllocate = true)
public double doubleValue( )
```

Allocates no memory. Does not allow "this" to escape local variables.

@SCJAllowed
@SCJRestricted({javax.safetycritical.annotate.Phase.ALL},
   maySelfSuspend = false,
   mayAllocate = true)
public boolean equals(Object obj)

Allocates no memory. Does not allow "this" or "obj" argument to escape local variables.

@SCJAllowed
@SCJRestricted({javax.safetycritical.annotate.Phase.ALL},
   maySelfSuspend = false,
   mayAllocate = true)
public static int floatToIntBits(**float** v)

Allocates no memory.

@SCJAllowed
@SCJRestricted({javax.safetycritical.annotate.Phase.ALL},
   maySelfSuspend = false,
   mayAllocate = true)
public static int floatToRawIntBits(**float** v)

Allocates no memory.

@SCJAllowed
@SCJRestricted({javax.safetycritical.annotate.Phase.ALL},
   maySelfSuspend = false,
   mayAllocate = true)
public float floatValue( )

Allocates no memory. Does not allow "this" to escape local variables.

@SCJAllowed
@SCJRestricted({javax.safetycritical.annotate.Phase.ALL},
   maySelfSuspend = false,
   mayAllocate = true)
public int hashCode( )

Allocates no memory. Does not allow "this" to escape local variables.

```
@SCJAllowed
@SCJRestricted({javax.safetycritical.annotate.Phase.ALL},
    maySelfSuspend = false,
    mayAllocate = true)
public static float intBitsToFloat(int v)
```

Allocates no memory.

```
@SCJAllowed
@SCJRestricted({javax.safetycritical.annotate.Phase.ALL},
    maySelfSuspend = false,
    mayAllocate = true)
public int intValue( )
```

Allocates no memory. Does not allow "this" to escape local variables.

```
@SCJAllowed
@SCJRestricted({javax.safetycritical.annotate.Phase.ALL},
    maySelfSuspend = false,
    mayAllocate = true)
public static boolean isInfinite(float v)
```

Allocates no memory.

```
@SCJAllowed
@SCJRestricted({javax.safetycritical.annotate.Phase.ALL},
    maySelfSuspend = false,
    mayAllocate = true)
public boolean isInfinite( )
```

Allocates no memory. Does not allow "this" to escape local variables.

```
@SCJAllowed
@SCJRestricted({javax.safetycritical.annotate.Phase.ALL},
    maySelfSuspend = false,
    mayAllocate = true)
public static boolean isNaN(float v)
```

Allocates no memory.

```
@SCJAllowed
@SCJRestricted({javax.safetycritical.annotate.Phase.ALL},
    maySelfSuspend = false,
    mayAllocate = true)
public boolean isNaN( )
```

Allocates no memory. Does not allow "this" to escape local variables.

```
@SCJAllowed
@SCJRestricted({javax.safetycritical.annotate.Phase.ALL},
   maySelfSuspend = false,
   mayAllocate = true)
```
public long longValue( )

Allocates no memory. Does not allow "this" to escape local variables.

```
@SCJAllowed
@SCJRestricted({javax.safetycritical.annotate.Phase.ALL},
   maySelfSuspend = false,
   mayAllocate = true)
```
public static float parseFloat(String s)
  throws java.lang.NumberFormatException

Allocates no memory. Does not allow "s" argument to escape local variables.

```
@SCJAllowed
@SCJRestricted({javax.safetycritical.annotate.Phase.ALL},
   maySelfSuspend = false,
   mayAllocate = true)
```
public short shortValue( )

Allocates no memory. Does not allow "this" to escape local variables.

```
@SCJAllowed
@SCJRestricted({javax.safetycritical.annotate.Phase.ALL},
   maySelfSuspend = false,
   mayAllocate = true)
```
public static java.lang.String toHexString(**float** v)

Allocates a String and associated internal "structure" (e.g. char[]) in caller's scope.

Memory behavior: This constructor may allocate objects within the currently active MemoryArea.

```
@SCJAllowed
@SCJRestricted({javax.safetycritical.annotate.Phase.ALL},
    maySelfSuspend = false,
    mayAllocate = true)
public static java.lang.String toString(float v)
```

Allocates a String and associated internal "structure" (e.g. char[]) in caller's scope.

Memory behavior: This constructor may allocate objects within the currently active MemoryArea.

```
@SCJAllowed
@SCJRestricted({javax.safetycritical.annotate.Phase.ALL},
    maySelfSuspend = false,
    mayAllocate = true)
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()).

Memory behavior: This constructor may allocate objects within the currently active MemoryArea.

```
@SCJAllowed
@SCJRestricted({javax.safetycritical.annotate.Phase.ALL},
    maySelfSuspend = false,
    mayAllocate = true)
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.

```
@SCJAllowed
@SCJRestricted({javax.safetycritical.annotate.Phase.ALL},
   maySelfSuspend = false,
   mayAllocate = true)
public static java.lang.Float valueOf(float val)
```

Allocates a Float in caller's scope.

Memory behavior: This constructor may allocate objects within the currently active MemoryArea.

B.2.18 CLASS **IllegalArgumentException**

```
@SCJAllowed
public class IllegalArgumentException
  implements java.io.Serializable
  extends java.lang.RuntimeException
```

Constructors

```
@SCJAllowed @SCJRestricted(maySelfSuspend = false)
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).

Memory behavior: This constructor may allocate objects within the currently active MemoryArea.

```
@SCJAllowed @SCJRestricted(maySelfSuspend = false)
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).

Memory behavior: This constructor may allocate objects within the currently active MemoryArea.

This constructor requires that the "msg" argument reside in a scope that encloses the scope of the "this" argument.

@SCJAllowed @SCJRestricted(maySelfSuspend = false)
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).

Memory behavior: This constructor may allocate objects within the currently active MemoryArea.

This constructor requires that the "msg" argument reside in a scope that encloses the scope of the "this" argument. This constructor requires that the "t" argument reside in a scope that encloses the scope of the "this" argument.

@SCJAllowed @SCJRestricted(maySelfSuspend = false)
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).

Memory behavior: This constructor may allocate objects within the currently active MemoryArea.

This constructor requires that the "t" argument reside in a scope that encloses the scope of the "this" argument.

B.2.19 CLASS **IllegalMonitorStateException**

@SCJAllowed(javax.safetycritical.annotate.Level.LEVEL_2)
public class IllegalMonitorStateException
  implements java.io.Serializable
  extends java.lang.RuntimeException

Constructors

@SCJAllowed @SCJRestricted(maySelfSuspend = false)
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).

Memory behavior: This constructor may allocate objects within the currently active MemoryArea.

@SCJAllowed @SCJRestricted(maySelfSuspend = false)
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 Illegal-AssignmentError will be thrown.

Allocates an application- and implementation-dependent amount of memory in the current scope (to represent stack backtrace).

Memory behavior: This constructor may allocate objects within the currently active MemoryArea.

This constructor requires that the "msg" argument reside in a scope that encloses the scope of the "this" argument.

@SCJAllowed @SCJRestricted(maySelfSuspend = false)
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).

Memory behavior: This constructor may allocate objects within the currently active MemoryArea.

This constructor requires that the "msg" argument reside in a scope that encloses the scope of the "this" argument. This constructor requires that the "t" argument reside in a scope that encloses the scope of the "this" argument.

@SCJAllowed @SCJRestricted(maySelfSuspend = false)
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).

Memory behavior: This constructor may allocate objects within the currently active MemoryArea.

This constructor requires that the "t" argument reside in a scope that encloses the scope of the "this" argument.

B.2.20 CLASS IllegalStateException

@SCJAllowed
public class IllegalStateException
  implements java.io.Serializable
  extends java.lang.RuntimeException

Constructors

```
@SCJAllowed
@SCJRestricted({javax.safetycritical.annotate.Phase.ALL},
   maySelfSuspend = false,
   mayAllocate = true)
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).

Memory behavior: This constructor may allocate objects within the currently active MemoryArea.

```
@SCJAllowed
@SCJRestricted({javax.safetycritical.annotate.Phase.ALL},
   maySelfSuspend = false,
   mayAllocate = true)
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).

Memory behavior: This constructor may allocate objects within the currently active MemoryArea.

This constructor requires that the "msg" argument reside in a scope that encloses the scope of the "this" argument.

B.2.21 CLASS IllegalThreadStateException

```
@SCJAllowed
public class IllegalThreadStateException
  implements java.io.Serializable
  extends java.lang.RuntimeException
```

Constructors

```
@SCJAllowed
@SCJRestricted({javax.safetycritical.annotate.Phase.ALL},
   maySelfSuspend = false,
   mayAllocate = true)
public IllegalThreadStateException( )
```

Memory behavior: This constructor may allocate objects within the currently active MemoryArea.

```
@SCJAllowed
@SCJRestricted({javax.safetycritical.annotate.Phase.ALL},
   maySelfSuspend = false,
   mayAllocate = true)
public IllegalThreadStateException(String description)
```

Memory behavior: This constructor may allocate objects within the currently active MemoryArea.

This constructor requires that the "msg" argument reside in a scope that encloses the scope of the "this" argument.

B.2.22 CLASS IncompatibleClassChangeError

```
@SCJAllowed(javax.safetycritical.annotate.Level.LEVEL_2)
public class IncompatibleClassChangeError
  implements java.io.Serializable
  extends java.lang.RuntimeException
```

Constructors

```
@SCJAllowed
@SCJRestricted({javax.safetycritical.annotate.Phase.ALL},
   maySelfSuspend = false,
   mayAllocate = true)
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).

Memory behavior: This constructor may allocate objects within the currently active MemoryArea.

```
@SCJAllowed
@SCJRestricted({javax.safetycritical.annotate.Phase.ALL},
   maySelfSuspend = false,
   mayAllocate = true)
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).

Memory behavior: This constructor may allocate objects within the currently active MemoryArea.

This constructor requires that the "msg" argument reside in a scope that encloses the scope of the "this" argument.

B.2.23 CLASS IndexOutOfBoundsException

```
@SCJAllowed
public class IndexOutOfBoundsException
  implements java.io.Serializable
  extends java.lang.RuntimeException
```

Constructors

```
@SCJAllowed @SCJRestricted(maySelfSuspend = false)
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).

Memory behavior: This constructor may allocate objects within the currently active MemoryArea.

```
@SCJAllowed @SCJRestricted(maySelfSuspend = false)
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).

Memory behavior: This constructor may allocate objects within the currently active MemoryArea.

This constructor requires that the "msg" argument reside in a scope that encloses the scope of the "this" argument.

B.2.24 CLASS **InstantiationException**

```
@SCJAllowed
public class InstantiationException
  implements java.io.Serializable
  extends java.lang.Exception
```

Constructors

```
@SCJAllowed
@SCJRestricted({javax.safetycritical.annotate.Phase.ALL},
   maySelfSuspend = false,
   mayAllocate = true)
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).

Memory behavior: This constructor may allocate objects within the currently active MemoryArea.

```
@SCJAllowed
@SCJRestricted({javax.safetycritical.annotate.Phase.ALL},
   maySelfSuspend = false,
   mayAllocate = true)
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 Illegal-AssignmentError will be thrown.

Allocates an application- and implementation-dependent amount of memory in the current scope (to represent stack backtrace).

Memory behavior: This constructor may allocate objects within the currently active MemoryArea.

This constructor requires that the "msg" argument reside in a scope that encloses the scope of the "this" argument.

B.2.25 CLASS **Integer**

```
@SCJAllowed
public class Integer
  implements java.lang.Comparable, java.io.Serializable
  extends java.lang.Number
```

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 @SCJRestricted(maySelfSuspend = false)
public Integer(**int** val)

Allocates no memory. Does not allow "this" to escape local variables.

@SCJAllowed @SCJRestricted(maySelfSuspend = false)
public Integer(String str)
  throws java.lang.NumberFormatException

Allocates no memory. Does not allow "this" or "str" argument to escape local variables.

Methods

@SCJAllowed @SCJRestricted(maySelfSuspend = false)
public static int bitCount(**int** i)

Allocates no memory. Does not allow "this" to escape local variables.

@SCJAllowed @SCJRestricted(maySelfSuspend = false)
public byte byteValue( )

Allocates no memory. Does not allow "this" to escape local variables.

@SCJAllowed @SCJRestricted(maySelfSuspend = false)
public int compareTo(Integer other)

Allocates no memory. Does not allow "this" or "other" argument to escape local variables.

@SCJAllowed @SCJRestricted(maySelfSuspend = false)
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 @SCJRestricted(maySelfSuspend = false)
public double doubleValue( )

Allocates no memory.

@SCJAllowed @SCJRestricted(maySelfSuspend = false)
public boolean equals(Object obj)

Allocates no memory. Does not allow "this" or "obj" argument to escape local variables.

@SCJAllowed @SCJRestricted(maySelfSuspend = false)
public float floatValue( )

Allocates no memory. Does not allow "this" to escape local variables.

@SCJAllowed @SCJRestricted(maySelfSuspend = false)
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.

Memory behavior: This constructor may allocate objects within the currently active MemoryArea.

@SCJAllowed @SCJRestricted(maySelfSuspend = false)
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.

Memory behavior: This constructor may allocate objects within the currently active MemoryArea.

@SCJAllowed @SCJRestricted(maySelfSuspend = false)
public static java.lang.Integer getInteger(String str)

Does not allow "str" argument to escape local variables. Allocates Integer in caller's scope.

Memory behavior: This constructor may allocate objects within the currently active MemoryArea.

@SCJAllowed @SCJRestricted(maySelfSuspend = false)
public int hashCode( )

Allocates no memory. Does not allow "this" to escape local variables.

@SCJAllowed @SCJRestricted(maySelfSuspend = false)
public static int highestOneBit(**int** i)

Allocates no memory.

@SCJAllowed @SCJRestricted(maySelfSuspend = false)
public int intValue( )

Allocates no memory. Does not allow "this" to escape local variables.

@SCJAllowed @SCJRestricted(maySelfSuspend = false)
public long longValue( )

Allocates no memory. Does not allow "this" to escape local variables.

@SCJAllowed @SCJRestricted(maySelfSuspend = false)
public static int lowestOneBit(**int** i)

Allocates no memory.

```
@SCJAllowed @SCJRestricted(maySelfSuspend = false)
public static int numberOfLeadingZeros(int i)
```

Allocates no memory.

```
@SCJAllowed @SCJRestricted(maySelfSuspend = false)
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 @SCJRestricted(maySelfSuspend = false)
public static int parseInt(String str)
  throws java.lang.NumberFormatException
```

Allocates no memory. Does not allow "str" argument to escape local variables.

```
@SCJAllowed @SCJRestricted(maySelfSuspend = false)
public static int reverse(int i)
```

Allocates no memory.

```
@SCJAllowed @SCJRestricted(maySelfSuspend = false)
public static int reverseBytes(int i)
```

Allocates no memory.

```
@SCJAllowed @SCJRestricted(maySelfSuspend = false)
public static int rotateLeft(int i, int distance)
```

Allocates no memory.

```
@SCJAllowed @SCJRestricted(maySelfSuspend = false)
public static int rotateRight(int i, int distance)
```

Allocates no memory.

@SCJAllowed @SCJRestricted(maySelfSuspend = false)
public short shortValue( )

Allocates no memory. Does not allow "this" to escape local variables.

@SCJAllowed @SCJRestricted(maySelfSuspend = false)
public static int sigNum(**int** i)

Allocates no memory.

@SCJAllowed @SCJRestricted(maySelfSuspend = false)
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.

Memory behavior: This constructor may allocate objects within the currently active MemoryArea.

@SCJAllowed @SCJRestricted(maySelfSuspend = false)
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.

Memory behavior: This constructor may allocate objects within the currently active MemoryArea.

@SCJAllowed @SCJRestricted(maySelfSuspend = false)
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.

Memory behavior: This constructor may allocate objects within the currently active MemoryArea.

@SCJAllowed @SCJRestricted(maySelfSuspend = false)
public static java.lang.String toString(**int** v)

Allocates a String and associated internal "structure" (e.g. char[]) in caller's scope.

Memory behavior: This constructor may allocate objects within the currently active MemoryArea.

@SCJAllowed @SCJRestricted(maySelfSuspend = false)
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.

Memory behavior: This constructor may allocate objects within the currently active MemoryArea.

@SCJAllowed @SCJRestricted(maySelfSuspend = false)
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()).

Memory behavior: This constructor may allocate objects within the currently active MemoryArea.

@SCJAllowed @SCJRestricted(maySelfSuspend = false)
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.

Memory behavior: This constructor may allocate objects within the currently active MemoryArea.

@SCJAllowed @SCJRestricted(maySelfSuspend = false)
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.

Memory behavior: This constructor may allocate objects within the currently active MemoryArea.

@SCJAllowed @SCJRestricted(maySelfSuspend = false)
public static java.lang.Integer valueOf(**int** val)

Allocates an Integer in caller's scope.

Memory behavior: This constructor may allocate objects within the currently active MemoryArea.

B.2.26 CLASS InternalError

@SCJAllowed
public class InternalError
  implements java.io.Serializable
  extends java.lang.VirtualMachineError

Constructors

```
@SCJAllowed
@SCJRestricted({javax.safetycritical.annotate.Phase.ALL},
   maySelfSuspend = false,
   mayAllocate = true)
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).

Memory behavior: This constructor may allocate objects within the currently active MemoryArea.

```
@SCJAllowed
@SCJRestricted({javax.safetycritical.annotate.Phase.ALL},
   maySelfSuspend = false,
   mayAllocate = true)
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 Illegal-AssignmentError will be thrown.

Allocates an application- and implementation-dependent amount of memory in the current scope (to represent stack backtrace).

Memory behavior: This constructor may allocate objects within the currently active MemoryArea.

This constructor requires that the "msg" argument reside in a scope that encloses the scope of the "this" argument.

B.2.27 CLASS **InterruptedException**

@SCJAllowed
public class InterruptedException
  implements java.io.Serializable
  extends java.lang.Exception

Constructors

@SCJAllowed
@SCJRestricted({javax.safetycritical.annotate.Phase.ALL},
  maySelfSuspend = false,
  mayAllocate = true)
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).

Memory behavior: This constructor may allocate objects within the currently active MemoryArea.

@SCJAllowed
@SCJRestricted({javax.safetycritical.annotate.Phase.ALL},
  maySelfSuspend = false,
  mayAllocate = true)
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).

Memory behavior: This constructor may allocate objects within the currently active MemoryArea.

This constructor requires that the "msg" argument reside in a scope that encloses the scope of the "this" argument.

B.2.28 CLASS **Long**

@SCJAllowed
public class Long
  implements java.lang.Comparable, java.io.Serializable
  extends java.lang.Number

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 @SCJRestricted(maySelfSuspend = false)
public Long(**long** val)

Allocates no memory. Does not allow "this" to escape local variables.

@SCJAllowed @SCJRestricted(maySelfSuspend = false)
public Long(String str)
  throws java.lang.NumberFormatException

Allocates no memory. Does not allow "this" or "str" argument to escape local variables.

Methods

@SCJAllowed @SCJRestricted(maySelfSuspend = false, mayAllocate = false)
public static int bitCount(**long** i)

Allocates no memory.

@SCJAllowed @SCJRestricted(maySelfSuspend = false, mayAllocate = false)
public byte byteValue( )

Allocates no memory. Does not allow "this" to escape local variables.

@SCJAllowed @SCJRestricted(maySelfSuspend = false, mayAllocate = false)
public int compareTo(**Long** other)

Allocates no memory. Does not allow "this" or "other" argument to escape local variables.

@SCJAllowed @SCJRestricted(maySelfSuspend = false)
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.

@SCJAllowed @SCJRestricted(maySelfSuspend = false, mayAllocate = false)
public double doubleValue( )

Allocates no memory. Does not allow "this" to escape local variables.

@SCJAllowed @SCJRestricted(maySelfSuspend = false, mayAllocate = false)
public boolean equals(Object obj)

Allocates no memory. Does not allow "this" or "obj" argument to escape local variables.

@SCJAllowed @SCJRestricted(maySelfSuspend = false, mayAllocate = false)
public float floatValue( )

Allocates no memory. Does not allow "this" to escape local variables.

@SCJAllowed @SCJRestricted(maySelfSuspend = false)
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 @SCJRestricted(maySelfSuspend = false)
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 @SCJRestricted(maySelfSuspend = false)
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 @SCJRestricted(maySelfSuspend = false, mayAllocate = false)
public int hashCode( )

Allocates no memory. Does not allow "this" to escape local variables.

@SCJAllowed @SCJRestricted(maySelfSuspend = false, mayAllocate = false)
public static long highestOneBit(**long** i)

Allocates no memory.

@SCJAllowed @SCJRestricted(maySelfSuspend = false, mayAllocate = false)
public int intValue( )

Allocates no memory. Does not allow "this" to escape local variables.

@SCJAllowed @SCJRestricted(maySelfSuspend = false, mayAllocate = false)
public long longValue( )

Allocates no memory. Does not allow "this" to escape local variables.

@SCJAllowed @SCJRestricted(maySelfSuspend = false, mayAllocate = false)
public static long lowestOneBit(**long** i)

Allocates no memory.

@SCJAllowed @SCJRestricted(maySelfSuspend = false, mayAllocate = false)
public static int numberOfLeadingZeros(**long** i)

Allocates no memory.

@SCJAllowed @SCJRestricted(maySelfSuspend = false, mayAllocate = false)
public static int numberOfTrailingZeros(**long** i)

Allocates no memory.

@SCJAllowed @SCJRestricted(maySelfSuspend = false, mayAllocate = false)
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 @SCJRestricted(maySelfSuspend = false, mayAllocate = false)
public static long parseLong(String str)
  throws java.lang.NumberFormatException

Allocates no memory. Does not allow "this" or "other" argument to escape local variables.

@SCJAllowed @SCJRestricted(maySelfSuspend = false, mayAllocate = false)
public static long reverse(**long** i)

Allocates no memory.

@SCJAllowed @SCJRestricted(maySelfSuspend = false, mayAllocate = false)
public static long reverseBytes(**long** i)

Allocates no memory.

@SCJAllowed @SCJRestricted(maySelfSuspend = false, mayAllocate = false)
public static long rotateLeft(**long** i, **int** distance)

Allocates no memory.

@SCJAllowed @SCJRestricted(maySelfSuspend = false, mayAllocate = false)
public static long rotateRight(**long** i, **int** distance)

Allocates no memory.

@SCJAllowed @SCJRestricted(maySelfSuspend = false, mayAllocate = false)
public short shortValue( )

Allocates no memory. Does not allow "this" to escape local variables.

@SCJAllowed @SCJRestricted(maySelfSuspend = false, mayAllocate = false)
public static int signum(**long** i)

Allocates no memory.

@SCJAllowed @SCJRestricted(maySelfSuspend = false)
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.

Memory behavior: This constructor may allocate objects within the currently active MemoryArea.

@SCJAllowed @SCJRestricted(maySelfSuspend = false)
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.

Memory behavior: This constructor may allocate objects within the currently active MemoryArea.

@SCJAllowed @SCJRestricted(maySelfSuspend = false)
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.

Memory behavior: This constructor may allocate objects within the currently active MemoryArea.

@SCJAllowed @SCJRestricted(maySelfSuspend = false)
public static java.lang.String toString(**long** v)

Allocates a String and associated internal "structure" (e.g. char[]) in caller's scope.

Memory behavior: This constructor may allocate objects within the currently active MemoryArea.

@SCJAllowed @SCJRestricted(maySelfSuspend = false)
public static java.lang.String toString(**long** v, **int** base)

Allocates a String and associated internal "structure" (e.g. char[]) in caller's scope.

Memory behavior: This constructor may allocate objects within the currently active MemoryArea.

@SCJAllowed @SCJRestricted(maySelfSuspend = false)
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()).

Memory behavior: This constructor may allocate objects within the currently active MemoryArea.

@SCJAllowed @SCJRestricted(maySelfSuspend = false)
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.

Memory behavior: This constructor may allocate objects within the currently active MemoryArea.

@SCJAllowed @SCJRestricted(maySelfSuspend = false)
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.

Memory behavior: This constructor may allocate objects within the currently active MemoryArea.

@SCJAllowed @SCJRestricted(maySelfSuspend = false)
public static java.lang.**Long** valueOf(**long** val)

Allocates a Long in caller's scope.

Memory behavior: This constructor may allocate objects within the currently active MemoryArea.

B.2.29 CLASS Math

@SCJAllowed
public final class Math **extends** java.lang.Object

Fields

@SCJAllowed
public static final double E

@SCJAllowed
public static final double PI

Constructors

@SCJAllowed
public Math( )

Methods

@SCJAllowed
@SCJRestricted({javax.safetycritical.annotate.Phase.ALL},
  maySelfSuspend = false,
  mayAllocate = true)
public static double IEEEremainder(**double** f1, **double** f2)

Allocates no memory.

@SCJAllowed
@SCJRestricted({javax.safetycritical.annotate.Phase.ALL},
  maySelfSuspend = false,
  mayAllocate = true)
public static long abs(**long** a)

Allocates no memory.

@SCJAllowed
@SCJRestricted({javax.safetycritical.annotate.Phase.ALL},
  maySelfSuspend = false,
  mayAllocate = true)
public static double abs(**double** a)

Allocates no memory.

@SCJAllowed
@SCJRestricted({javax.safetycritical.annotate.Phase.ALL},
  maySelfSuspend = false,
  mayAllocate = true)
public static float abs(**float** a)

Allocates no memory.

@SCJAllowed
@SCJRestricted({javax.safetycritical.annotate.Phase.ALL},
    maySelfSuspend = false,
    mayAllocate = true)
public static int abs(**int** a)

Allocates no memory.

@SCJAllowed
@SCJRestricted({javax.safetycritical.annotate.Phase.ALL},
    maySelfSuspend = false,
    mayAllocate = true)
public static double acos(**double** a)

Allocates no memory.

@SCJAllowed
@SCJRestricted({javax.safetycritical.annotate.Phase.ALL},
    maySelfSuspend = false,
    mayAllocate = true)
public static double asin(**double** a)

Allocates no memory.

@SCJAllowed
@SCJRestricted({javax.safetycritical.annotate.Phase.ALL},
    maySelfSuspend = false,
    mayAllocate = true)
public static double atan(**double** a)

Allocates no memory.

@SCJAllowed
@SCJRestricted({javax.safetycritical.annotate.Phase.ALL},
    maySelfSuspend = false,
    mayAllocate = true)
public static double atan2(**double** a, **double** b)

Allocates no memory.

@SCJAllowed
@SCJRestricted({javax.safetycritical.annotate.Phase.ALL},
    maySelfSuspend = false,
    mayAllocate = true)
public static double cbrt(**double** a)

Allocates no memory.

@SCJAllowed
@SCJRestricted({javax.safetycritical.annotate.Phase.ALL},
    maySelfSuspend = false,
    mayAllocate = true)
public static double ceil(**double** a)

Allocates no memory.

@SCJAllowed
@SCJRestricted({javax.safetycritical.annotate.Phase.ALL},
    maySelfSuspend = false,
    mayAllocate = true)
public static double copySign(**float** magnitude, **float** sign)

Allocates no memory.

@SCJAllowed
@SCJRestricted({javax.safetycritical.annotate.Phase.ALL},
    maySelfSuspend = false,
    mayAllocate = true)
public static double copySign(**double** magnitude, **double** sign)

Allocates no memory.

@SCJAllowed
@SCJRestricted({javax.safetycritical.annotate.Phase.ALL},
    maySelfSuspend = false,
    mayAllocate = true)
public static double cos(**double** a)

Allocates no memory.

@SCJAllowed
@SCJRestricted({javax.safetycritical.annotate.Phase.ALL},
    maySelfSuspend = false,
    mayAllocate = true)
public static double cosh(**double** a)

Allocates no memory.

```
@SCJAllowed
@SCJRestricted({javax.safetycritical.annotate.Phase.ALL},
    maySelfSuspend = false,
    mayAllocate = true)
```
public static double exp(**double** a)

Allocates no memory.

```
@SCJAllowed
@SCJRestricted({javax.safetycritical.annotate.Phase.ALL},
    maySelfSuspend = false,
    mayAllocate = true)
```
public static double expm1(**double** a)

Allocates no memory.

```
@SCJAllowed
@SCJRestricted({javax.safetycritical.annotate.Phase.ALL},
    maySelfSuspend = false,
    mayAllocate = true)
```
public static double floor(**double** a)

Allocates no memory.

```
@SCJAllowed
@SCJRestricted({javax.safetycritical.annotate.Phase.ALL},
    maySelfSuspend = false,
    mayAllocate = true)
```
public static int getExponent(**float** a)

Allocates no memory.

```
@SCJAllowed
@SCJRestricted({javax.safetycritical.annotate.Phase.ALL},
    maySelfSuspend = false,
    mayAllocate = true)
```
public static int getExponent(**double** a)

Allocates no memory.

@SCJAllowed
@SCJRestricted({javax.safetycritical.annotate.Phase.ALL},
    maySelfSuspend = false,
    mayAllocate = true)
public static double hypot(**double** x, **double** y)

Allocates no memory.

@SCJAllowed
@SCJRestricted({javax.safetycritical.annotate.Phase.ALL},
    maySelfSuspend = false,
    mayAllocate = true)
public static double log(**double** a)

Allocates no memory.

@SCJAllowed
@SCJRestricted({javax.safetycritical.annotate.Phase.ALL},
    maySelfSuspend = false,
    mayAllocate = true)
public static double log10(**double** a)

Allocates no memory.

@SCJAllowed
@SCJRestricted({javax.safetycritical.annotate.Phase.ALL},
    maySelfSuspend = false,
    mayAllocate = true)
public static double log1p(**double** a)

Allocates no memory.

@SCJAllowed
@SCJRestricted({javax.safetycritical.annotate.Phase.ALL},
    maySelfSuspend = false,
    mayAllocate = true)
public static double max(**double** a, **double** b)

Allocates no memory.

@SCJAllowed
@SCJRestricted({javax.safetycritical.annotate.Phase.ALL},
    maySelfSuspend = false,
    mayAllocate = true)
public static long max(**long** a, **long** b)

Allocates no memory.

@SCJAllowed
@SCJRestricted({javax.safetycritical.annotate.Phase.ALL},
    maySelfSuspend = false,
    mayAllocate = true)
public static float max(**float** a, **float** b)

Allocates no memory.

@SCJAllowed
@SCJRestricted({javax.safetycritical.annotate.Phase.ALL},
    maySelfSuspend = false,
    mayAllocate = true)
public static int max(**int** a, **int** b)

Allocates no memory.

@SCJAllowed
@SCJRestricted({javax.safetycritical.annotate.Phase.ALL},
    maySelfSuspend = false,
    mayAllocate = true)
public static long min(**long** a, **long** b)

Allocates no memory.

@SCJAllowed
@SCJRestricted({javax.safetycritical.annotate.Phase.ALL},
    maySelfSuspend = false,
    mayAllocate = true)
public static double min(**double** a, **double** b)

Allocates no memory.

@SCJAllowed
@SCJRestricted({javax.safetycritical.annotate.Phase.ALL},
    maySelfSuspend = false,
    mayAllocate = true)
public static float min(**float** a, **float** b)

Allocates no memory.

@SCJAllowed
@SCJRestricted({javax.safetycritical.annotate.Phase.ALL},
    maySelfSuspend = false,
    mayAllocate = true)
public static int min(**int** a, **int** b)

Allocates no memory.

@SCJAllowed
@SCJRestricted({javax.safetycritical.annotate.Phase.ALL},
    maySelfSuspend = false,
    mayAllocate = true)
public static float nextAfter(**float** start, **float** direction)

Allocates no memory.

@SCJAllowed
@SCJRestricted({javax.safetycritical.annotate.Phase.ALL},
    maySelfSuspend = false,
    mayAllocate = true)
public static double nextAfter(**double** start, **double** direction)

Allocates no memory.

@SCJAllowed
@SCJRestricted({javax.safetycritical.annotate.Phase.ALL},
    maySelfSuspend = false,
    mayAllocate = true)
public static float nextUp(**float** d)

Allocates no memory.

@SCJAllowed
@SCJRestricted({javax.safetycritical.annotate.Phase.ALL},
    maySelfSuspend = false,
    mayAllocate = true)
public static double nextUp(**double** d)

Allocates no memory.

@SCJAllowed
@SCJRestricted({javax.safetycritical.annotate.Phase.ALL},
    maySelfSuspend = false,
    mayAllocate = true)
public static double pow(**double** a, **double** b)

Allocates no memory.

@SCJAllowed
@SCJRestricted({javax.safetycritical.annotate.Phase.ALL},
 maySelfSuspend = false,
 mayAllocate = true)
public static double random( )

Allocates no memory.

@SCJAllowed
@SCJRestricted({javax.safetycritical.annotate.Phase.ALL},
 maySelfSuspend = false,
 mayAllocate = true)
public static double rint(**double** a)

Allocates no memory.

@SCJAllowed
@SCJRestricted({javax.safetycritical.annotate.Phase.ALL},
 maySelfSuspend = false,
 mayAllocate = true)
public static long round(**double** a)

Allocates no memory.

@SCJAllowed
@SCJRestricted({javax.safetycritical.annotate.Phase.ALL},
 maySelfSuspend = false,
 mayAllocate = true)
public static int round(**float** a)

Allocates no memory.

@SCJAllowed
@SCJRestricted({javax.safetycritical.annotate.Phase.ALL},
 maySelfSuspend = false,
 mayAllocate = true)
public static float scalb(**float** f, **int** scaleFactor)

Allocates no memory.

```
@SCJAllowed
@SCJRestricted({javax.safetycritical.annotate.Phase.ALL},
   maySelfSuspend = false,
   mayAllocate = true)
```
public static double scalb(**double** d, **int** scaleFactor)

Allocates no memory.

```
@SCJAllowed
@SCJRestricted({javax.safetycritical.annotate.Phase.ALL},
   maySelfSuspend = false,
   mayAllocate = true)
```
public static float signum(**float** f)

Allocates no memory.

```
@SCJAllowed
@SCJRestricted({javax.safetycritical.annotate.Phase.ALL},
   maySelfSuspend = false,
   mayAllocate = true)
```
public static double signum(**double** d)

Allocates no memory.

```
@SCJAllowed
@SCJRestricted({javax.safetycritical.annotate.Phase.ALL},
   maySelfSuspend = false,
   mayAllocate = true)
```
public static double sin(**double** a)

Allocates no memory.

```
@SCJAllowed
@SCJRestricted({javax.safetycritical.annotate.Phase.ALL},
   maySelfSuspend = false,
   mayAllocate = true)
```
public static double sinh(**double** a)

Allocates no memory.

```
@SCJAllowed
@SCJRestricted({javax.safetycritical.annotate.Phase.ALL},
   maySelfSuspend = false,
   mayAllocate = true)
```
public static double sqrt(**double** a)

Allocates no memory.

```
@SCJAllowed
@SCJRestricted({javax.safetycritical.annotate.Phase.ALL},
   maySelfSuspend = false,
   mayAllocate = true)
```
public static double tan(**double** a)

Allocates no memory.

```
@SCJAllowed
@SCJRestricted({javax.safetycritical.annotate.Phase.ALL},
   maySelfSuspend = false,
   mayAllocate = true)
```
public static double tanh(**double** a)

Allocates no memory.

```
@SCJAllowed
@SCJRestricted({javax.safetycritical.annotate.Phase.ALL},
   maySelfSuspend = false,
   mayAllocate = true)
```
public static double toDegrees(**double** val)

Allocates no memory.

```
@SCJAllowed
@SCJRestricted({javax.safetycritical.annotate.Phase.ALL},
   maySelfSuspend = false,
   mayAllocate = true)
```
public static double toRadians(**double** val)

Allocates no memory.

```
@SCJAllowed
@SCJRestricted({javax.safetycritical.annotate.Phase.ALL},
   maySelfSuspend = false,
   mayAllocate = true)
```
public static float ulp(**float** d)

Allocates no memory.

@SCJAllowed
@SCJRestricted({javax.safetycritical.annotate.Phase.ALL},
    maySelfSuspend = false,
    mayAllocate = true)
public static double ulp(**double** d)

Allocates no memory.

B.2.30 CLASS NegativeArraySizeException

@SCJAllowed
public class NegativeArraySizeException
  implements java.io.Serializable
  extends java.lang.RuntimeException

Constructors

@SCJAllowed @SCJRestricted(maySelfSuspend = false)
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).

Memory behavior: This constructor may allocate objects within the currently active MemoryArea.

@SCJAllowed @SCJRestricted(maySelfSuspend = false)
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 Illegal-AssignmentError will be thrown.

Allocates an application- and implementation-dependent amount of memory in the current scope (to represent stack backtrace).

Memory behavior: This constructor may allocate objects within the currently active MemoryArea.

This constructor requires that the "msg" argument reside in a scope that encloses the scope of the "this" argument.

B.2.31 CLASS **NullPointerException**

```
@SCJAllowed
public class NullPointerException
  implements java.io.Serializable
  extends java.lang.RuntimeException
```

Constructors

```
@SCJAllowed
@SCJRestricted({javax.safetycritical.annotate.Phase.ALL},
   maySelfSuspend = false,
   mayAllocate = true)
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).

Memory behavior: This constructor may allocate objects within the currently active MemoryArea.

```
@SCJAllowed
@SCJRestricted({javax.safetycritical.annotate.Phase.ALL},
   maySelfSuspend = false,
   mayAllocate = true)
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 Illegal-AssignmentError will be thrown.

Allocates an application- and implementation-dependent amount of memory in the current scope (to represent stack backtrace).

Memory behavior: This constructor may allocate objects within the currently active MemoryArea.

This constructor requires that the "msg" argument reside in a scope that encloses the scope of the "this" argument.

B.2.32 CLASS **Number**

@SCJAllowed
public abstract class Number
  implements java.io.Serializable
  extends java.lang.Object

Constructors

@SCJAllowed
@SCJRestricted({javax.safetycritical.annotate.Phase.ALL},
   maySelfSuspend = false,
   mayAllocate = true)
public Number( )

The implementation of this method shall not allow "this" to escape the method's local variables.

Methods

@SCJAllowed
@SCJRestricted({javax.safetycritical.annotate.Phase.ALL},
   maySelfSuspend = false,
   mayAllocate = true)
public byte byteValue( )

The implementation of this method shall not allow "this" to escape the method's local variables.

@SCJAllowed
@SCJRestricted({javax.safetycritical.annotate.Phase.ALL},
   maySelfSuspend = false,
   mayAllocate = true)
public abstract double doubleValue( )

The implementation of this method shall not allow "this" to escape the method's local variables.

@SCJAllowed
@SCJRestricted({javax.safetycritical.annotate.Phase.ALL},
  maySelfSuspend = false,
  mayAllocate = true)
public abstract float floatValue( )

The implementation of this method shall not allow "this" to escape the method's local variables.

@SCJAllowed
@SCJRestricted({javax.safetycritical.annotate.Phase.ALL},
  maySelfSuspend = false,
  mayAllocate = true)
public abstract int intValue( )

The implementation of this method shall not allow "this" to escape the method's local variables.

@SCJAllowed
@SCJRestricted({javax.safetycritical.annotate.Phase.ALL},
  maySelfSuspend = false,
  mayAllocate = true)
public abstract long longValue( )

The implementation of this method shall not allow "this" to escape the method's local variables.

@SCJAllowed
@SCJRestricted({javax.safetycritical.annotate.Phase.ALL},
  maySelfSuspend = false,
  mayAllocate = true)
public abstract short shortValue( )

The implementation of this method shall not allow "this" to escape the method's local variables.

B.2.33 CLASS NumberFormatException

@SCJAllowed
public class NumberFormatException
  implements java.io.Serializable
  extends java.lang.IllegalArgumentException

Constructors

@SCJAllowed
@SCJRestricted({javax.safetycritical.annotate.Phase.ALL},
  maySelfSuspend = false,
  mayAllocate = true)
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).

Memory behavior: This constructor may allocate objects within the currently active MemoryArea.

@SCJAllowed
@SCJRestricted({javax.safetycritical.annotate.Phase.ALL},
  maySelfSuspend = false,
  mayAllocate = true)
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 Illegal-AssignmentError will be thrown.

Allocates an application- and implementation-dependent amount of memory in the current scope (to represent stack backtrace).

Memory behavior: This constructor may allocate objects within the currently active MemoryArea.

This constructor requires that the "msg" argument reside in a scope that encloses the scope of the "this" argument.

B.2.34 CLASS **Object**

@SCJAllowed
public class Object

Constructors

@SCJAllowed
@SCJRestricted({javax.safetycritical.annotate.Phase.ALL},
    maySelfSuspend = false,
    mayAllocate = true)
public Object( )

Allocates no memory. Does not allow "this" to escape local variables.

Methods

@SCJAllowed
@SCJRestricted({javax.safetycritical.annotate.Phase.ALL},
    maySelfSuspend = false,
    mayAllocate = true)
public boolean equals(Object obj)

Allocates no memory. Does not allow "this" or "obj" argument to escape local variables.

@SCJAllowed
@SCJRestricted({javax.safetycritical.annotate.Phase.ALL},
    maySelfSuspend = false,
    mayAllocate = false)
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
@SCJRestricted({javax.safetycritical.annotate.Phase.ALL},
    maySelfSuspend = false,
    mayAllocate = true)
public int hashCode( )

Allocates no memory. Does not allow "this" to escape local variables.

@SCJAllowed(javax.safetycritical.annotate.Level.LEVEL_2)
@SCJRestricted({javax.safetycritical.annotate.Phase.ALL},
    maySelfSuspend = false,
    mayAllocate = false)
public final void notify( )

Allocates no memory. Does not allow "this" to escape local variables.

@SCJAllowed(javax.safetycritical.annotate.Level.LEVEL_2)
@SCJRestricted({javax.safetycritical.annotate.Phase.ALL},
    maySelfSuspend = false,
    mayAllocate = false)
public final void notifyAll( )

Allocates no memory. Does not allow "this" to escape local variables.

@SCJAllowed
@SCJRestricted({javax.safetycritical.annotate.Phase.ALL},
    maySelfSuspend = false,
    mayAllocate = true)
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.

Memory behavior: This constructor may allocate objects within the currently active MemoryArea.

@SCJAllowed(javax.safetycritical.annotate.Level.LEVEL_2)
@SCJRestricted({javax.safetycritical.annotate.Phase.ALL},
    maySelfSuspend = true,
    mayAllocate = false)
public final void wait(**long** timeout, **int** nanos)
  throws java.lang.InterruptedException

Allocates no memory. Does not allow "this" to escape local variables.

@SCJAllowed(javax.safetycritical.annotate.Level.LEVEL_2)
@SCJRestricted({javax.safetycritical.annotate.Phase.ALL},
    maySelfSuspend = true,
    mayAllocate = false)
public final void wait(**long** timeout)
  throws java.lang.InterruptedException

Allocates no memory. Does not allow "this" to escape local variables.

@SCJAllowed(javax.safetycritical.annotate.Level.LEVEL_2)
@SCJRestricted({javax.safetycritical.annotate.Phase.ALL},
  maySelfSuspend = true,
  mayAllocate = false)
public final void wait( )

Allocates no memory. Does not allow "this" to escape local variables.

B.2.35 CLASS **OutOfMemoryError**

@SCJAllowed
public class OutOfMemoryError
 implements java.io.Serializable
 extends java.lang.VirtualMachineError

Constructors

@SCJAllowed
@SCJRestricted({javax.safetycritical.annotate.Phase.ALL},
  maySelfSuspend = false,
  mayAllocate = true)
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).

Memory behavior: This constructor may allocate objects within the currently active MemoryArea.

@SCJAllowed
@SCJRestricted({javax.safetycritical.annotate.Phase.ALL},
  maySelfSuspend = false,
  mayAllocate = true)
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 Illegal-AssignmentError will be thrown.

Allocates an application- and implementation-dependent amount of memory in the current scope (to represent stack backtrace).

Memory behavior: This constructor may allocate objects within the currently active MemoryArea.

This constructor requires that the "msg" argument reside in a scope that encloses the scope of the "this" argument.

B.2.36 CLASS **RuntimeException**

@SCJAllowed
public class RuntimeException
  implements java.io.Serializable
  extends java.lang.Exception

Constructors

@SCJAllowed @SCJRestricted(maySelfSuspend = false)
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).

Memory behavior: This constructor may allocate objects within the currently active MemoryArea.

@SCJAllowed @SCJRestricted(maySelfSuspend = false)
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 Illegal-AssignmentError will be thrown.

Allocates an application- and implementation-dependent amount of memory in the current scope (to represent stack backtrace).

Memory behavior: This constructor may allocate objects within the currently active MemoryArea.

This constructor requires that the "msg" argument reside in a scope that encloses the scope of the "this" argument.

```
@SCJAllowed @SCJRestricted(maySelfSuspend = false)
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).

Memory behavior: This constructor may allocate objects within the currently active MemoryArea.

This constructor requires that the "msg" argument reside in a scope that encloses the scope of the "this" argument. This constructor requires that the "t" argument reside in a scope that encloses the scope of the "this" argument.

```
@SCJAllowed @SCJRestricted(maySelfSuspend = false)
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).

Memory behavior: This constructor may allocate objects within the currently active MemoryArea.

This constructor requires that the "t" argument reside in a scope that encloses the scope of the "this" argument.

B.2.37 CLASS **Short**

@SCJAllowed
public class Short
  implements java.lang.Comparable, java.io.Serializable
  extends java.lang.Number

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 @SCJRestricted(maySelfSuspend = false)
public Short(**short** val)

Allocates no memory. Does not allow "this" to escape local variables.

@SCJAllowed @SCJRestricted(maySelfSuspend = false)
public Short(String str)
  throws java.lang.NumberFormatException

Allocates no memory. Does not allow "this" or "str" argument to escape local variables.

Methods

@SCJAllowed @SCJRestricted(maySelfSuspend = false, mayAllocate = false)
public byte byteValue( )

Allocates no memory. Does not allow "this" to escape local variables.

@SCJAllowed @SCJRestricted(maySelfSuspend = false, mayAllocate = false)
public int compareTo(**Short** other)

Allocates no memory. Does not allow "this" or "other" argument to escape local variables.

@SCJAllowed @SCJRestricted(maySelfSuspend = false)
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.

@SCJAllowed @SCJRestricted(maySelfSuspend = false, mayAllocate = false)
public double doubleValue( )

Allocates no memory.

@SCJAllowed @SCJRestricted(maySelfSuspend = false, mayAllocate = false)
public boolean equals(Object obj)

Allocates no memory. Does not allow "this" or "obj" argument to escape local variables.

@SCJAllowed @SCJRestricted(maySelfSuspend = false)
public float floatValue( )

Allocates no memory. Does not allow "this" to escape local variables.

@SCJAllowed @SCJRestricted(maySelfSuspend = false)
public int hashCode( )

Allocates no memory. Does not allow "this" to escape local variables.

@SCJAllowed @SCJRestricted(maySelfSuspend = false, mayAllocate = false)
public int intValue( )

Allocates no memory. Does not allow "this" to escape local variables.

@SCJAllowed @SCJRestricted(maySelfSuspend = false, mayAllocate = false)
public long longValue( )

Allocates no memory. Does not allow "this" to escape local variables.

@SCJAllowed @SCJRestricted(maySelfSuspend = false, mayAllocate = false)
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 @SCJRestricted(maySelfSuspend = false, mayAllocate = false)
public static short parseShort(String str)
  throws java.lang.NumberFormatException

Allocates no memory. Does not allow "str" argument to escape local variables.

@SCJAllowed @SCJRestricted(maySelfSuspend = false, mayAllocate = false)
public static short reverseBytes(**short** i)

Allocates no memory.

@SCJAllowed @SCJRestricted(maySelfSuspend = false, mayAllocate = false)
public short shortValue( )

Allocates no memory. Does not allow "this" to escape local variables.

@SCJAllowed @SCJRestricted(maySelfSuspend = false)
public static java.lang.String toString(**short** v)

Allocates a String and associated internal "structure" (e.g. char[]) in caller's scope.

Memory behavior: This constructor may allocate objects within the currently active MemoryArea.

@SCJAllowed @SCJRestricted(maySelfSuspend = false)
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()).

Memory behavior: This constructor may allocate objects within the currently active MemoryArea.

@SCJAllowed @SCJRestricted(maySelfSuspend = false)
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.

Memory behavior: This constructor may allocate objects within the currently active MemoryArea.

@SCJAllowed @SCJRestricted(maySelfSuspend = false)
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.

Memory behavior: This constructor may allocate objects within the currently active MemoryArea.

@SCJAllowed @SCJRestricted(maySelfSuspend = false)
public static java.lang.**Short** valueOf(**short** val)

Allocates a Short in caller's scope.

Memory behavior: This constructor may allocate objects within the currently active MemoryArea.

B.2.38 CLASS StackOverflowError

@SCJAllowed
public class StackOverflowError
  implements java.io.Serializable
  extends java.lang.VirtualMachineError

Constructors

@SCJAllowed @SCJRestricted(maySelfSuspend = false)
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).

Memory behavior: This constructor may allocate objects within the currently active MemoryArea.

@SCJAllowed @SCJRestricted(maySelfSuspend = false)
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 Illegal-AssignmentError will be thrown.

Allocates an application- and implementation-dependent amount of memory in the current scope (to represent stack backtrace).

Memory behavior: This constructor may allocate objects within the currently active MemoryArea.

This constructor requires that the "msg" argument reside in a scope that encloses the scope of the "this" argument.

B.2.39 CLASS StackTraceElement

@SCJAllowed
public class StackTraceElement **extends** java.lang.Object

Constructors

@SCJAllowed @SCJRestricted(maySelfSuspend = false)
public StackTraceElement(String declaringClass,
  String methodName,
  String fileName,
  int lineNumber)

Shall not copy "this" to any instance or static field.

Memory behavior: This constructor requires that the "declaringClass" argument reside in a scope that encloses the scope of the "this" argument. This constructor requires that the "methodName" argument reside in a scope that encloses the scope of the "this" argument. This constructor requires that the "fileName" argument reside in a scope that encloses the scope of the "this" argument.

Methods

@SCJAllowed @SCJRestricted(maySelfSuspend = false)
public boolean equals(Object obj)

Allocates no memory. Does not allow "this" or "obj" argument to escape local variables.

@SCJAllowed @SCJRestricted(maySelfSuspend = false)
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 @SCJRestricted(maySelfSuspend = false)
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 @SCJRestricted(maySelfSuspend = false)
public int getLineNumber( )

Performs no memory allocation.

@SCJAllowed @SCJRestricted(maySelfSuspend = false)
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 @SCJRestricted(maySelfSuspend = false)
public int hashCode( )

Allocates no memory. Does not allow "this" to escape local variables.

@SCJAllowed @SCJRestricted(maySelfSuspend = false)
public boolean isNativeMethod( )

Allocates no memory. Does not allow "this" to escape local variables.

@SCJAllowed @SCJRestricted(maySelfSuspend = false)
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()).

Memory behavior: This constructor may allocate objects within the currently active MemoryArea.

B.2.40 CLASS **StrictMath**

@SCJAllowed
public final class StrictMath **extends** java.lang.Object

Fields

@SCJAllowed
public static final double E

@SCJAllowed
public static final double PI

Methods

@SCJAllowed
@SCJRestricted({javax.safetycritical.annotate.Phase.ALL},
  maySelfSuspend = false,
  mayAllocate = true)
public static double IEEEremainder(**double** f1, **double** f2)

Allocates no memory.

@SCJAllowed
@SCJRestricted({javax.safetycritical.annotate.Phase.ALL},
  maySelfSuspend = false,
  mayAllocate = true)
public static long abs(**long** a)

Allocates no memory.

@SCJAllowed
@SCJRestricted({javax.safetycritical.annotate.Phase.ALL},
  maySelfSuspend = false,
  mayAllocate = true)
public static double abs(**double** a)

Allocates no memory.

@SCJAllowed
@SCJRestricted({javax.safetycritical.annotate.Phase.ALL},
  maySelfSuspend = false,
  mayAllocate = true)
public static float abs(**float** a)

Allocates no memory.

```
@SCJAllowed
@SCJRestricted({javax.safetycritical.annotate.Phase.ALL},
   maySelfSuspend = false,
   mayAllocate = true)
public static int abs(int a)
```

Allocates no memory.

```
@SCJAllowed
@SCJRestricted({javax.safetycritical.annotate.Phase.ALL},
   maySelfSuspend = false,
   mayAllocate = true)
public static double acos(double a)
```

Allocates no memory.

```
@SCJAllowed
@SCJRestricted({javax.safetycritical.annotate.Phase.ALL},
   maySelfSuspend = false,
   mayAllocate = true)
public static double asin(double a)
```

Allocates no memory.

```
@SCJAllowed
@SCJRestricted({javax.safetycritical.annotate.Phase.ALL},
   maySelfSuspend = false,
   mayAllocate = true)
public static double atan(double a)
```

Allocates no memory.

```
@SCJAllowed
@SCJRestricted({javax.safetycritical.annotate.Phase.ALL},
   maySelfSuspend = false,
   mayAllocate = true)
public static double atan2(double a, double b)
```

Allocates no memory.

@SCJAllowed
@SCJRestricted({javax.safetycritical.annotate.Phase.ALL},
    maySelfSuspend = false,
    mayAllocate = true)
public static double cbrt(**double** a)

Allocates no memory.

@SCJAllowed
@SCJRestricted({javax.safetycritical.annotate.Phase.ALL},
    maySelfSuspend = false,
    mayAllocate = true)
public static double ceil(**double** a)

Allocates no memory.

@SCJAllowed
@SCJRestricted({javax.safetycritical.annotate.Phase.ALL},
    maySelfSuspend = false,
    mayAllocate = true)
public static double copySign(**float** magnitude, **float** sign)

Allocates no memory.

@SCJAllowed
@SCJRestricted({javax.safetycritical.annotate.Phase.ALL},
    maySelfSuspend = false,
    mayAllocate = true)
public static double copySign(**double** magnitude, **double** sign)

Allocates no memory.

@SCJAllowed
@SCJRestricted({javax.safetycritical.annotate.Phase.ALL},
    maySelfSuspend = false,
    mayAllocate = true)
public static double cos(**double** a)

Allocates no memory.

@SCJAllowed
@SCJRestricted({javax.safetycritical.annotate.Phase.ALL},
    maySelfSuspend = false,
    mayAllocate = true)
public static double cosh(**double** a)

Allocates no memory.

```
@SCJAllowed
@SCJRestricted({javax.safetycritical.annotate.Phase.ALL},
   maySelfSuspend = false,
   mayAllocate = true)
```
public static double exp(**double** a)

Allocates no memory.

```
@SCJAllowed
@SCJRestricted({javax.safetycritical.annotate.Phase.ALL},
   maySelfSuspend = false,
   mayAllocate = true)
```
public static double expm1(**double** a)

Allocates no memory.

```
@SCJAllowed
@SCJRestricted({javax.safetycritical.annotate.Phase.ALL},
   maySelfSuspend = false,
   mayAllocate = true)
```
public static double floor(**double** a)

Allocates no memory.

```
@SCJAllowed
@SCJRestricted({javax.safetycritical.annotate.Phase.ALL},
   maySelfSuspend = false,
   mayAllocate = true)
```
public static int getExponent(**float** a)

Allocates no memory.

```
@SCJAllowed
@SCJRestricted({javax.safetycritical.annotate.Phase.ALL},
   maySelfSuspend = false,
   mayAllocate = true)
```
public static int getExponent(**double** a)

Allocates no memory.

@SCJAllowed
@SCJRestricted({javax.safetycritical.annotate.Phase.ALL},
    maySelfSuspend = false,
    mayAllocate = true)
public static double hypot(**double** x, **double** y)

Allocates no memory.

@SCJAllowed
@SCJRestricted({javax.safetycritical.annotate.Phase.ALL},
    maySelfSuspend = false,
    mayAllocate = true)
public static double log(**double** a)

Allocates no memory.

@SCJAllowed
@SCJRestricted({javax.safetycritical.annotate.Phase.ALL},
    maySelfSuspend = false,
    mayAllocate = true)
public static double log10(**double** a)

Allocates no memory.

@SCJAllowed
@SCJRestricted({javax.safetycritical.annotate.Phase.ALL},
    maySelfSuspend = false,
    mayAllocate = true)
public static double log1p(**double** a)

Allocates no memory.

@SCJAllowed
@SCJRestricted({javax.safetycritical.annotate.Phase.ALL},
    maySelfSuspend = false,
    mayAllocate = true)
public static double max(**double** a, **double** b)

Allocates no memory.

@SCJAllowed
@SCJRestricted({javax.safetycritical.annotate.Phase.ALL},
    maySelfSuspend = false,
    mayAllocate = true)
public static long max(**long** a, **long** b)

Allocates no memory.

```
@SCJAllowed
@SCJRestricted({javax.safetycritical.annotate.Phase.ALL},
   maySelfSuspend = false,
   mayAllocate = true)
```
public static float max(**float** a, **float** b)

Allocates no memory.

```
@SCJAllowed
@SCJRestricted({javax.safetycritical.annotate.Phase.ALL},
   maySelfSuspend = false,
   mayAllocate = true)
```
public static int max(**int** a, **int** b)

Allocates no memory.

```
@SCJAllowed
@SCJRestricted({javax.safetycritical.annotate.Phase.ALL},
   maySelfSuspend = false,
   mayAllocate = true)
```
public static long min(**long** a, **long** b)

Allocates no memory.

```
@SCJAllowed
@SCJRestricted({javax.safetycritical.annotate.Phase.ALL},
   maySelfSuspend = false,
   mayAllocate = true)
```
public static double min(**double** a, **double** b)

Allocates no memory.

```
@SCJAllowed
@SCJRestricted({javax.safetycritical.annotate.Phase.ALL},
   maySelfSuspend = false,
   mayAllocate = true)
```
public static float min(**float** a, **float** b)

Allocates no memory.

@SCJAllowed
@SCJRestricted({javax.safetycritical.annotate.Phase.ALL},
    maySelfSuspend = false,
    mayAllocate = true)
public static int min(**int** a, **int** b)

Allocates no memory.

@SCJAllowed
@SCJRestricted({javax.safetycritical.annotate.Phase.ALL},
    maySelfSuspend = false,
    mayAllocate = true)
public static float nextAfter(**float** start, **float** direction)

Allocates no memory.

@SCJAllowed
@SCJRestricted({javax.safetycritical.annotate.Phase.ALL},
    maySelfSuspend = false,
    mayAllocate = true)
public static double nextAfter(**double** start, **double** direction)

Allocates no memory.

@SCJAllowed
@SCJRestricted({javax.safetycritical.annotate.Phase.ALL},
    maySelfSuspend = false,
    mayAllocate = true)
public static float nextUp(**float** d)

Allocates no memory.

@SCJAllowed
@SCJRestricted({javax.safetycritical.annotate.Phase.ALL},
    maySelfSuspend = false,
    mayAllocate = true)
public static double nextUp(**double** d)

Allocates no memory.

@SCJAllowed
@SCJRestricted({javax.safetycritical.annotate.Phase.ALL},
    maySelfSuspend = false,
    mayAllocate = true)
public static double pow(**double** a, **double** b)

Allocates no memory.

@SCJAllowed
@SCJRestricted({javax.safetycritical.annotate.Phase.ALL},
    maySelfSuspend = false,
    mayAllocate = true)
public static double random( )

Allocates no memory.

@SCJAllowed
@SCJRestricted({javax.safetycritical.annotate.Phase.ALL},
    maySelfSuspend = false,
    mayAllocate = true)
public static double rint(**double** a)

Allocates no memory.

@SCJAllowed
@SCJRestricted({javax.safetycritical.annotate.Phase.ALL},
    maySelfSuspend = false,
    mayAllocate = true)
public static long round(**double** a)

Allocates no memory.

@SCJAllowed
@SCJRestricted({javax.safetycritical.annotate.Phase.ALL},
    maySelfSuspend = false,
    mayAllocate = true)
public static int round(**float** a)

Allocates no memory.

@SCJAllowed
@SCJRestricted({javax.safetycritical.annotate.Phase.ALL},
    maySelfSuspend = false,
    mayAllocate = true)
public static float scalb(**float** f, **int** scaleFactor)

Allocates no memory.

```
@SCJAllowed
@SCJRestricted({javax.safetycritical.annotate.Phase.ALL},
   maySelfSuspend = false,
   mayAllocate = true)
public static double scalb(double d, int scaleFactor)
```

Allocates no memory.

```
@SCJAllowed
@SCJRestricted({javax.safetycritical.annotate.Phase.ALL},
   maySelfSuspend = false,
   mayAllocate = true)
public static float signum(float f)
```

Allocates no memory.

```
@SCJAllowed
@SCJRestricted({javax.safetycritical.annotate.Phase.ALL},
   maySelfSuspend = false,
   mayAllocate = true)
public static double signum(double d)
```

Allocates no memory.

```
@SCJAllowed
@SCJRestricted({javax.safetycritical.annotate.Phase.ALL},
   maySelfSuspend = false,
   mayAllocate = true)
public static double sin(double a)
```

Allocates no memory.

```
@SCJAllowed
@SCJRestricted({javax.safetycritical.annotate.Phase.ALL},
   maySelfSuspend = false,
   mayAllocate = true)
public static double sinh(double a)
```

Allocates no memory.

```
@SCJAllowed
@SCJRestricted({javax.safetycritical.annotate.Phase.ALL},
   maySelfSuspend = false,
   mayAllocate = true)
public static double sqrt(double a)
```

Allocates no memory.

```
@SCJAllowed
@SCJRestricted({javax.safetycritical.annotate.Phase.ALL},
   maySelfSuspend = false,
   mayAllocate = true)
public static double tan(double a)
```

Allocates no memory.

```
@SCJAllowed
@SCJRestricted({javax.safetycritical.annotate.Phase.ALL},
   maySelfSuspend = false,
   mayAllocate = true)
public static double tanh(double a)
```

Allocates no memory.

```
@SCJAllowed
@SCJRestricted({javax.safetycritical.annotate.Phase.ALL},
   maySelfSuspend = false,
   mayAllocate = true)
public static double toDegrees(double val)
```

Allocates no memory.

```
@SCJAllowed
@SCJRestricted({javax.safetycritical.annotate.Phase.ALL},
   maySelfSuspend = false,
   mayAllocate = true)
public static double toRadians(double val)
```

Allocates no memory.

```
@SCJAllowed
@SCJRestricted({javax.safetycritical.annotate.Phase.ALL},
   maySelfSuspend = false,
   mayAllocate = true)
public static float ulp(float d)
```

Allocates no memory.

@SCJAllowed
@SCJRestricted({javax.safetycritical.annotate.Phase.ALL},
  maySelfSuspend = false,
  mayAllocate = true)
public static double ulp(**double** d)

Allocates no memory.

B.2.41 CLASS String

@SCJAllowed
public final class String
  implements java.lang.CharSequence, java.lang.Comparable, java.io.Serializable
  extends java.lang.Object

Constructors

@SCJAllowed
@SCJRestricted({javax.safetycritical.annotate.Phase.ALL},
  maySelfSuspend = false,
  mayAllocate = true)
public String( )

Does not allow "this" argument to escape local variables.

@SCJAllowed
@SCJRestricted({javax.safetycritical.annotate.Phase.ALL},
  maySelfSuspend = false,
  mayAllocate = true)
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".

Memory behavior: This constructor may allocate objects within the same MemoryArea that holds the implicit this argument.

@SCJAllowed
@SCJRestricted({javax.safetycritical.annotate.Phase.ALL},
  maySelfSuspend = false,
  mayAllocate = true)
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".

Memory behavior: This constructor may allocate objects within the same MemoryArea that holds the implicit this argument.

```
@SCJAllowed
@SCJRestricted({javax.safetycritical.annotate.Phase.ALL},
   maySelfSuspend = false,
   mayAllocate = true)
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".

Memory behavior: This constructor may allocate objects within the same MemoryArea that holds the implicit this argument.

```
@SCJAllowed
@SCJRestricted({javax.safetycritical.annotate.Phase.ALL},
   maySelfSuspend = false,
   mayAllocate = true)
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".

Memory behavior: This constructor may allocate objects within the same MemoryArea that holds the implicit this argument.

```
@SCJAllowed
@SCJRestricted({javax.safetycritical.annotate.Phase.ALL},
   maySelfSuspend = false,
   mayAllocate = true)
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.

Memory behavior: This constructor requires that the "b" argument reside in a scope that encloses the scope of the "this" argument.

```
@SCJAllowed
@SCJRestricted({javax.safetycritical.annotate.Phase.ALL},
   maySelfSuspend = false,
   mayAllocate = true)
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".

Memory behavior: This constructor may allocate objects within the same MemoryArea that holds the implicit this argument.

Methods

```
@SCJAllowed
@SCJRestricted({javax.safetycritical.annotate.Phase.ALL},
   maySelfSuspend = false,
   mayAllocate = true)
public char charAt(int index)
```

Allocates no memory. Does not allow "this" to escape local variables.

```
@SCJAllowed
@SCJRestricted({javax.safetycritical.annotate.Phase.ALL},
   maySelfSuspend = false,
   mayAllocate = true)
public int compareTo(String str)
```

Allocates no memory. Does not allow "this" or "str" argument to escape local variables.

```
@SCJAllowed
@SCJRestricted({javax.safetycritical.annotate.Phase.ALL},
   maySelfSuspend = false,
   mayAllocate = true)
```
public int compareToIgnoreCase(String str)

Allocates no memory. Does not allow "this" or "str" argument to escape local variables.

```
@SCJAllowed
@SCJRestricted({javax.safetycritical.annotate.Phase.ALL},
   maySelfSuspend = false,
   mayAllocate = true)
```
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.

Memory behavior: This constructor may allocate objects within the currently active MemoryArea.

```
@SCJAllowed
@SCJRestricted({javax.safetycritical.annotate.Phase.ALL},
   maySelfSuspend = false,
   mayAllocate = true)
```
public boolean contains(CharSequence arg)

Does not allow "this" or "str" argument to escape local variables.

```
@SCJAllowed
@SCJRestricted({javax.safetycritical.annotate.Phase.ALL},
   maySelfSuspend = false,
   mayAllocate = true)
```
public boolean contentEquals(CharSequence cs)

Does not allow "this" or "str" argument to escape local variables.

```
@SCJAllowed
@SCJRestricted({javax.safetycritical.annotate.Phase.ALL},
    maySelfSuspend = false,
    mayAllocate = false)
```
public final boolean endsWith(String suffix)

Allocates no memory. Does not allow "this" or "suffix" argument to escape local variables.

```
@SCJAllowed
@SCJRestricted({javax.safetycritical.annotate.Phase.ALL},
    maySelfSuspend = false,
    mayAllocate = false)
```
public boolean equals(Object obj)

Allocates no memory. Does not allow "this" or "obj" argument to escape local variables.

```
@SCJAllowed
@SCJRestricted({javax.safetycritical.annotate.Phase.ALL},
    maySelfSuspend = false,
    mayAllocate = false)
```
public boolean equalsIgnoreCase(String str)

Allocates no memory. Does not allow "this" or "str" argument to escape local variables.

```
@SCJAllowed
@SCJRestricted({javax.safetycritical.annotate.Phase.ALL},
    maySelfSuspend = false,
    mayAllocate = true)
```
public byte[] getBytes( )

Does not allow "this" to escape local variables. Allocates a byte array in the caller's context.

Memory behavior: This constructor may allocate objects within the currently active MemoryArea.

```
@SCJAllowed
@SCJRestricted({javax.safetycritical.annotate.Phase.ALL},
   maySelfSuspend = false,
   mayAllocate = false)
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
@SCJRestricted({javax.safetycritical.annotate.Phase.ALL},
   maySelfSuspend = false,
   mayAllocate = false)
public int hashCode( )
```

Allocates no memory. Does not allow "this" to escape local variables.

```
@SCJAllowed
@SCJRestricted({javax.safetycritical.annotate.Phase.ALL},
   maySelfSuspend = false,
   mayAllocate = false)
public int indexOf(int ch, int from_index)
```

Allocates no memory. Does not allow "this" to escape local variables.

```
@SCJAllowed
@SCJRestricted({javax.safetycritical.annotate.Phase.ALL},
   maySelfSuspend = false,
   mayAllocate = false)
public int indexOf(String str, int from_index)
```

Allocates no memory. Does not allow "this" or "str" argument to escape local variables.

```
@SCJAllowed
@SCJRestricted({javax.safetycritical.annotate.Phase.ALL},
   maySelfSuspend = false,
   mayAllocate = false)
public int indexOf(String str)
```

Allocates no memory. Does not allow "this" or "str" argument to escape local variables.

```
@SCJAllowed
@SCJRestricted({javax.safetycritical.annotate.Phase.ALL},
   maySelfSuspend = false,
   mayAllocate = false)
public int indexOf(int ch)
```

Allocates no memory. Does not allow "this" to escape local variables.

```
@SCJAllowed
@SCJRestricted({javax.safetycritical.annotate.Phase.ALL},
   maySelfSuspend = false,
   mayAllocate = false)
public boolean isEmpty( )
```

Allocates no memory. Does not allow "this" argument to escape local variables.

```
@SCJAllowed
@SCJRestricted({javax.safetycritical.annotate.Phase.ALL},
   maySelfSuspend = false,
   mayAllocate = false)
public int lastIndexOf(String str)
```

Allocates no memory. Does not allow "this" or "str" argument to escape local variables.

```
@SCJAllowed
@SCJRestricted({javax.safetycritical.annotate.Phase.ALL},
   maySelfSuspend = false,
   mayAllocate = false)
public int lastIndexOf(String str, int from_index)
```

Allocates no memory. Does not allow "this" or "str" argument to escape local variables.

```
@SCJAllowed
@SCJRestricted({javax.safetycritical.annotate.Phase.ALL},
   maySelfSuspend = false,
   mayAllocate = false)
public int lastIndexOf(int ch, int from_index)
```

Allocates no memory. Does not allow "this" to escape local variables.

```
@SCJAllowed
@SCJRestricted({javax.safetycritical.annotate.Phase.ALL},
   maySelfSuspend = false,
   mayAllocate = false)
public int lastIndexOf(int ch)
```

Allocates no memory. Does not allow "this" to escape local variables.

```
@SCJAllowed
@SCJRestricted({javax.safetycritical.annotate.Phase.ALL},
   maySelfSuspend = false,
   mayAllocate = false)
public int length( )
```

Allocates no memory. Does not allow "this" to escape local variables.

```
@SCJAllowed
@SCJRestricted({javax.safetycritical.annotate.Phase.ALL},
   maySelfSuspend = false,
   mayAllocate = false)
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
@SCJRestricted({javax.safetycritical.annotate.Phase.ALL},
   maySelfSuspend = false,
   mayAllocate = false)
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
@SCJRestricted({javax.safetycritical.annotate.Phase.ALL},
   maySelfSuspend = false,
   mayAllocate = true)
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.

Memory behavior: This constructor may allocate objects within the currently active MemoryArea.

```
@SCJAllowed
@SCJRestricted({javax.safetycritical.annotate.Phase.ALL},
   maySelfSuspend = false,
   mayAllocate = true)
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.

Memory behavior: This constructor may allocate objects within the currently active MemoryArea.

```
@SCJAllowed
@SCJRestricted({javax.safetycritical.annotate.Phase.ALL},
   maySelfSuspend = false,
   mayAllocate = true)
public final boolean startsWith(String prefix, int toffset)
```

Allocates no memory. Does not allow "this" or "prefix" argument to escape local variables.

```
@SCJAllowed
@SCJRestricted({javax.safetycritical.annotate.Phase.ALL},
   maySelfSuspend = false,
   mayAllocate = false)
public final boolean startsWith(String prefix)
```

Allocates no memory. Does not allow "this" or "prefix" argument to escape local variables.

```
@SCJAllowed
@SCJRestricted({javax.safetycritical.annotate.Phase.ALL},
   maySelfSuspend = false,
   mayAllocate = true)
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.

Memory behavior: This constructor may allocate objects within the currently active MemoryArea.

This constructor requires that the "this" argument reside in a scope that encloses the scope of the "@result" argument.

```
@SCJAllowed
@SCJRestricted({javax.safetycritical.annotate.Phase.ALL},
   maySelfSuspend = false,
   mayAllocate = true)
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.

Memory behavior: This constructor may allocate objects within the currently active MemoryArea.

This constructor requires that the "this" argument reside in a scope that encloses the scope of the "@result" argument.

```
@SCJAllowed
@SCJRestricted({javax.safetycritical.annotate.Phase.ALL},
   maySelfSuspend = false,
   mayAllocate = true)
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.

Memory behavior: This constructor may allocate objects within the currently active MemoryArea.

This constructor requires that the "this" argument reside in a scope that encloses the scope of the "@result" argument.

```
@SCJAllowed
@SCJRestricted({javax.safetycritical.annotate.Phase.ALL},
   maySelfSuspend = false,
   mayAllocate = true)
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.

Memory behavior: This constructor may allocate objects within the currently active MemoryArea.

```
@SCJAllowed
@SCJRestricted({javax.safetycritical.annotate.Phase.ALL},
   maySelfSuspend = false,
   mayAllocate = true)
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.

Memory behavior: This constructor may allocate objects within the currently active MemoryArea.

@SCJAllowed
@SCJRestricted({javax.safetycritical.annotate.Phase.ALL},
    maySelfSuspend = false,
    mayAllocate = true)
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()).

Memory behavior: This constructor may allocate objects within the currently active MemoryArea.

@SCJAllowed
@SCJRestricted({javax.safetycritical.annotate.Phase.ALL},
    maySelfSuspend = false,
    mayAllocate = true)
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.

Memory behavior: This constructor may allocate objects within the currently active MemoryArea.

@SCJAllowed
@SCJRestricted({javax.safetycritical.annotate.Phase.ALL},
    maySelfSuspend = false,
    mayAllocate = true)
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.

Memory behavior: This constructor may allocate objects within the currently active MemoryArea.

This constructor requires that the "this" argument reside in a scope that encloses the scope of the "@result" argument.

@SCJAllowed
@SCJRestricted({javax.safetycritical.annotate.Phase.ALL},
    maySelfSuspend = false,
    mayAllocate = true)
public static java.lang.String valueOf(**float** f)

> Allocates a String and associated internal "structure" (e.g. char[]) in caller's scope.

Memory behavior: This constructor may allocate objects within the currently active MemoryArea.

@SCJAllowed
@SCJRestricted({javax.safetycritical.annotate.Phase.ALL},
    maySelfSuspend = false,
    mayAllocate = true)
public static java.lang.String valueOf(**int** i)

> Allocates a String and associated internal "structure" (e.g. char[]) in caller's scope.

Memory behavior: This constructor may allocate objects within the currently active MemoryArea.

@SCJAllowed
@SCJRestricted({javax.safetycritical.annotate.Phase.ALL},
    maySelfSuspend = false,
    mayAllocate = true)
public static java.lang.String valueOf(**long** l)

> Allocates a String and associated internal "structure" (e.g. char[]) in caller's scope.

Memory behavior: This constructor may allocate objects within the currently active MemoryArea.

```
@SCJAllowed
@SCJRestricted({javax.safetycritical.annotate.Phase.ALL},
   maySelfSuspend = false,
   mayAllocate = true)
public static java.lang.String valueOf(Object o)
```

Allocates a String object in the caller's scope.

Memory behavior: This constructor may allocate objects within the currently active MemoryArea.

```
@SCJAllowed
@SCJRestricted({javax.safetycritical.annotate.Phase.ALL},
   maySelfSuspend = false,
   mayAllocate = true)
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.

Memory behavior: This constructor may allocate objects within the currently active MemoryArea.

```
@SCJAllowed
@SCJRestricted({javax.safetycritical.annotate.Phase.ALL},
   maySelfSuspend = false,
   mayAllocate = true)
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.

Memory behavior: This constructor may allocate objects within the currently active MemoryArea.

```
@SCJAllowed
@SCJRestricted({javax.safetycritical.annotate.Phase.ALL},
   maySelfSuspend = false,
   mayAllocate = true)
public static java.lang.String valueOf(double d)
```

Allocates a String and associated internal "structure" (e.g. char[]) in caller's scope.

Memory behavior: This constructor may allocate objects within the currently active MemoryArea.

```
@SCJAllowed
@SCJRestricted({javax.safetycritical.annotate.Phase.ALL},
   maySelfSuspend = false,
   mayAllocate = true)
public static java.lang.String valueOf(char c)
```

Allocates a String and associated internal "structure" (e.g. char[]) in caller's scope.

Memory behavior: This constructor may allocate objects within the currently active MemoryArea.

```
@SCJAllowed
@SCJRestricted({javax.safetycritical.annotate.Phase.ALL},
   maySelfSuspend = false,
   mayAllocate = false)
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.2.42 CLASS **StringBuilder**

```
@SCJAllowed
public final class StringBuilder
  implements java.lang.Appendable, java.lang.CharSequence, java.io.Serializable
  extends java.lang.Object
```

Constructors

```
@SCJAllowed
@SCJRestricted({javax.safetycritical.annotate.Phase.ALL},
   maySelfSuspend = true,
   mayAllocate = true)
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".

Memory behavior: This constructor may allocate objects within the same MemoryArea that holds the implicit this argument.

```
@SCJAllowed
@SCJRestricted({javax.safetycritical.annotate.Phase.ALL},
   maySelfSuspend = true,
   mayAllocate = true)
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".

Memory behavior: This constructor may allocate objects within the same MemoryArea that holds the implicit this argument.

```
@SCJAllowed
@SCJRestricted({javax.safetycritical.annotate.Phase.ALL},
   maySelfSuspend = true,
   mayAllocate = true)
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".

Memory behavior: This constructor may allocate objects within the same MemoryArea that holds the implicit this argument.

```
@SCJAllowed
@SCJRestricted({javax.safetycritical.annotate.Phase.ALL},
   maySelfSuspend = true,
   mayAllocate = true)
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".

Memory behavior: This constructor may allocate objects within the same MemoryArea that holds the implicit this argument.

Methods

```
@SCJAllowed
@SCJRestricted({javax.safetycritical.annotate.Phase.ALL},
   maySelfSuspend = true,
   mayAllocate = true)
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.

Memory behavior: This constructor may allocate objects within the same MemoryArea that holds the implicit this argument.

```
@SCJAllowed
@SCJRestricted({javax.safetycritical.annotate.Phase.ALL},
   maySelfSuspend = true,
   mayAllocate = true)
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.

Memory behavior: This constructor may allocate objects within the same MemoryArea that holds the implicit this argument.

```
@SCJAllowed
@SCJRestricted({javax.safetycritical.annotate.Phase.ALL},
   maySelfSuspend = true,
   mayAllocate = true)
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.

Memory behavior: This constructor may allocate objects within the same MemoryArea that holds the implicit this argument.

```
@SCJAllowed
@SCJRestricted({javax.safetycritical.annotate.Phase.ALL},
   maySelfSuspend = true,
   mayAllocate = true)
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.

Memory behavior: This constructor may allocate objects within the same MemoryArea that holds the implicit this argument.

```
@SCJAllowed
@SCJRestricted({javax.safetycritical.annotate.Phase.ALL},
   maySelfSuspend = true,
   mayAllocate = true)
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.

Memory behavior: This constructor may allocate objects within the same MemoryArea that holds the implicit this argument.

```
@SCJAllowed
@SCJRestricted({javax.safetycritical.annotate.Phase.ALL},
   maySelfSuspend = true,
   mayAllocate = true)
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.

Memory behavior: This constructor may allocate objects within the same MemoryArea that holds the implicit this argument.

```
@SCJAllowed
@SCJRestricted({javax.safetycritical.annotate.Phase.ALL},
   maySelfSuspend = true,
   mayAllocate = true)
public java.lang.StringBuilder append(Object o)
```

Does not allow "this" to escape local variables. If expansion of "this" String-Builder'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"

Memory behavior: This constructor may allocate objects within the same MemoryArea that holds the implicit this argument.

```
@SCJAllowed
@SCJRestricted({javax.safetycritical.annotate.Phase.ALL},
   maySelfSuspend = true,
   mayAllocate = true)
public java.lang.StringBuilder append(int i)
```

Does not allow "this" to escape local variables. If expansion of "this" String-Builder'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.

Memory behavior: This constructor may allocate objects within the same MemoryArea that holds the implicit this argument.

```
@SCJAllowed
@SCJRestricted({javax.safetycritical.annotate.Phase.ALL},
   maySelfSuspend = true,
   mayAllocate = true)
public java.lang.StringBuilder append(double d)
```

Does not allow "this" to escape local variables. If expansion of "this" String-Builder'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.

Memory behavior: This constructor may allocate objects within the same MemoryArea that holds the implicit this argument.

```
@SCJAllowed
@SCJRestricted({javax.safetycritical.annotate.Phase.ALL},
   maySelfSuspend = true,
   mayAllocate = true)
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.

Memory behavior: This constructor may allocate objects within the same MemoryArea that holds the implicit this argument.

```
@SCJAllowed
@SCJRestricted({javax.safetycritical.annotate.Phase.ALL},
   maySelfSuspend = true,
   mayAllocate = true)
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.

Memory behavior: This constructor may allocate objects within the same MemoryArea that holds the implicit this argument.

```
@SCJAllowed
@SCJRestricted({javax.safetycritical.annotate.Phase.ALL},
   maySelfSuspend = true,
   mayAllocate = true)
public java.lang.StringBuilder append(boolean b)
```

Does not allow "this" to escape local variables. If expansion of "this" String-Builder'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.

Memory behavior: This constructor may allocate objects within the same MemoryArea that holds the implicit this argument.

```
@SCJAllowed
@SCJRestricted({javax.safetycritical.annotate.Phase.ALL},
   maySelfSuspend = true,
   mayAllocate = true)
public int capacity( )
```

Allocates no memory. Does not allow "this" to escape local variables.

```
@SCJAllowed
@SCJRestricted({javax.safetycritical.annotate.Phase.ALL},
   maySelfSuspend = true,
   mayAllocate = true)
public char charAt(int index)
```

Allocates no memory. Does not allow "this" to escape local variables.

```
@SCJAllowed
@SCJRestricted({javax.safetycritical.annotate.Phase.ALL},
   maySelfSuspend = true,
   mayAllocate = true)
public void ensureCapacity(int minimum_capacity)
```

Does not allow "this" to escape local variables. If expansion of "this" String-Builder'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.

Memory behavior: This constructor may allocate objects within the same MemoryArea that holds the implicit this argument.

```
@SCJAllowed
@SCJRestricted({javax.safetycritical.annotate.Phase.ALL},
    maySelfSuspend = true,
    mayAllocate = true)
public void getChars(int srcBegin,
  int srcEnd,
  char [] dst,
  int dstBegin)
```

Does not allow "this" or "dst" to escape local variables.

```
@SCJAllowed
@SCJRestricted({javax.safetycritical.annotate.Phase.ALL},
    maySelfSuspend = true,
    mayAllocate = true)
public void indexOf(String str, int fromIndex)
```

Does not allow "this" or "dst" to escape local variables.

```
@SCJAllowed
@SCJRestricted({javax.safetycritical.annotate.Phase.ALL},
    maySelfSuspend = true,
    mayAllocate = true)
public void indexOf(String str)
```

Does not allow "this" or "dst" to escape local variables.

```
@SCJAllowed
@SCJRestricted({javax.safetycritical.annotate.Phase.ALL},
    maySelfSuspend = true,
    mayAllocate = true)
public void lastIndexOf(String str, int fromIndex)
```

Does not allow "this" or "dst" to escape local variables.

```
@SCJAllowed
@SCJRestricted({javax.safetycritical.annotate.Phase.ALL},
    maySelfSuspend = true,
    mayAllocate = true)
public void lastIndexOf(String str)
```

Does not allow "this" or "dst" to escape local variables.

```
@SCJAllowed
@SCJRestricted({javax.safetycritical.annotate.Phase.ALL},
   maySelfSuspend = true,
   mayAllocate = true)
public int length( )
```

Allocates no memory. Does not allow "this" to escape local variables.

Memory behavior: This constructor may allocate objects within the same MemoryArea that holds the implicit this argument.

```
@SCJAllowed
@SCJRestricted({javax.safetycritical.annotate.Phase.ALL},
   maySelfSuspend = true,
   mayAllocate = true)
public void setLength(int new_length)
  throws java.lang.IndexOutOfBoundsException
```

Does not allow "this" to escape local variables.

```
@SCJAllowed
@SCJRestricted({javax.safetycritical.annotate.Phase.ALL},
   maySelfSuspend = true,
   mayAllocate = true)
public java.lang.CharSequence subSequence(int start, int end)
```

Does not allow "this" to escape local variables. Allocates a String in caller's scope.

Memory behavior: This constructor may allocate objects within the currently active MemoryArea.

```
@SCJAllowed
@SCJRestricted({javax.safetycritical.annotate.Phase.ALL},
   maySelfSuspend = true,
   mayAllocate = true)
public java.lang.String toString( )
```

Does not allow "this" to escape local variables. Allocates a String in caller's scope.

Memory behavior: This constructor may allocate objects within the currently active MemoryArea.

B.2.43 CLASS StringIndexOutOfBoundsException

@SCJAllowed
public class StringIndexOutOfBoundsException
  implements java.io.Serializable
  extends java.lang.IndexOutOfBoundsException

Constructors

@SCJAllowed
@SCJRestricted({javax.safetycritical.annotate.Phase.ALL},
    maySelfSuspend = false,
    mayAllocate = true)
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).

Memory behavior: This constructor may allocate objects within the currently active MemoryArea.

@SCJAllowed
@SCJRestricted({javax.safetycritical.annotate.Phase.ALL},
    maySelfSuspend = false,
    mayAllocate = true)
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).

Memory behavior: This constructor may allocate objects within the currently active MemoryArea.

```
@SCJAllowed
@SCJRestricted({javax.safetycritical.annotate.Phase.ALL},
    maySelfSuspend = false,
    mayAllocate = true)
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 Illegal-AssignmentError will be thrown.

Allocates an application- and implementation-dependent amount of memory in the current scope (to represent stack backtrace).

Memory behavior: This constructor may allocate objects within the currently active MemoryArea.

This constructor requires that the "msg" argument reside in a scope that encloses the scope of the "this" argument.

B.2.44 CLASS System

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

Constructors

```
@SCJAllowed
@SCJRestricted({javax.safetycritical.annotate.Phase.ALL},
    maySelfSuspend = false,
    mayAllocate = true)
protected System( )
```

Allocates no memory.

Methods

```
@SCJAllowed
@SCJRestricted({javax.safetycritical.annotate.Phase.ALL},
   maySelfSuspend = false,
   mayAllocate = true)
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
@SCJRestricted({javax.safetycritical.annotate.Phase.ALL},
   maySelfSuspend = false,
   mayAllocate = true)
public static long currentTimeMillis( )
```

Allocates no memory.

```
@SCJAllowed
@SCJRestricted({javax.safetycritical.annotate.Phase.ALL},
   maySelfSuspend = false,
   mayAllocate = true)
public static void exit(int code)
```

Allocates no memory.

```
@SCJAllowed
@SCJRestricted({javax.safetycritical.annotate.Phase.ALL},
   maySelfSuspend = false,
   mayAllocate = true)
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
@SCJRestricted({javax.safetycritical.annotate.Phase.ALL},
   maySelfSuspend = false,
   mayAllocate = true)
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
@SCJRestricted({javax.safetycritical.annotate.Phase.ALL},
   maySelfSuspend = false,
   mayAllocate = true)
public static int identityHashCode(Object x)
```

Does not allow argument "x" to escape local variables. Allocates no memory.

```
@SCJAllowed
@SCJRestricted({javax.safetycritical.annotate.Phase.ALL},
   maySelfSuspend = false,
   mayAllocate = true)
public static long nanoTime( )
```

Allocates no memory.

B.2.45 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 the 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)
@SCJRestricted({javax.safetycritical.annotate.Phase.ALL},
    maySelfSuspend = false,
    mayAllocate = true)
public static
java.lang.Thread.UncaughtExceptionHandler getDefaultUncaughtExceptionHandler( )
```

returns the default handler for uncaught exceptions.

Memory behavior: 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)
@SCJRestricted({javax.safetycritical.annotate.Phase.ALL},
    maySelfSuspend = false,
    mayAllocate = true)
public
java.lang.Thread.UncaughtExceptionHandler getUncaughtExceptionHandler( )
```

returns the handler invoked when this thread abruptly terminates due to an uncaught exception.

Memory behavior: 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)
@SCJRestricted({javax.safetycritical.annotate.Phase.ALL},
    maySelfSuspend = false,
    mayAllocate = true)
public void interrupt( )
```

Interrupts this thread.

Memory behavior: Allocates no memory. Does not allow this to escape local variables.

```
@SCJAllowed(javax.safetycritical.annotate.Level.LEVEL_2)
@SCJRestricted(maySelfSuspend = false)
public static boolean interrupted( )
```

Tests whether the current thread has been interrupted. The interrupted status of the thread is cleared by this method.

returns true if the current thread has been interrupted.

Memory behavior: Allocates no memory. Does not allow this to escape local variables.

```
@SCJAllowed(javax.safetycritical.annotate.Level.LEVEL_2)
@SCJRestricted({javax.safetycritical.annotate.Phase.ALL},
   maySelfSuspend = false,
   mayAllocate = true)
```
public final boolean isAlive( )

returns true if the current thread has not returned from run().

Memory behavior: Allocates no memory. Does not allow this to escape local variables.

```
@SCJAllowed(javax.safetycritical.annotate.Level.LEVEL_2)
@SCJRestricted({javax.safetycritical.annotate.Phase.ALL},
   maySelfSuspend = false,
   mayAllocate = true)
```
public boolean isInterrupted( )

Tests whether this thread has been interrupted. The interrupted status of the thread is not affected by this method.

returns true if the current thread has been interrupted.

Memory behavior: Allocates no memory. Does not allow this to escape local variables.

```
@SCJAllowed(javax.safetycritical.annotate.Level.SUPPORT)
```
public void run( )

Wait for completion of the thread, but do not wait longer than millis milliseconds and nanos nanoseconds. This method can be overridden to provide the code of the thread at Level 2. @memory Does not allow this to escape local variables.

```
@SCJRestricted({javax.safetycritical.annotate.Phase.ALL},
   maySelfSuspend = false,
   mayAllocate = true)
@SCJAllowed(javax.safetycritical.annotate.Level.LEVEL_2)
public static void setDefaultUncaughtExceptionHandler(
  Thread.UncaughtExceptionHandler eh)
```

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.

eh — is the default handler to be set.

Memory behavior: Allocates no memory. Does not allow this to escape local variables. The eh argument must reside in immortal memory.

Memory behavior: This constructor requires that the "eh" argument reside in a scope that encloses the scope of the "@immortal" argument.

```
@SCJAllowed(javax.safetycritical.annotate.Level.LEVEL_2)
@SCJRestricted({javax.safetycritical.annotate.Phase.ALL},
   maySelfSuspend = false,
   mayAllocate = false)
public void setUncaughtExceptionHandler(
  Thread.UncaughtExceptionHandler eh)
```

eh — the UncaughtExceptionHandler to be set for this thread.

Memory behavior: Allocates no memory. Does not allow this to escape local variables. The eh argument must reside in a scope that encloses the scope of this.

Memory behavior: This constructor requires that the "eh" argument reside in a scope that encloses the scope of the "this" argument.

```
@SCJRestricted({javax.safetycritical.annotate.Phase.ALL},
   maySelfSuspend = false,
   mayAllocate = true)
@SCJAllowed
public java.lang.String toString( )
```

returns a string representation of this thread, including the thread's name and priority.

Memory behavior: Does not allow this to escape local variables. Allocates a String and associated internal "structure" (e.g. char[]) in caller's scope.

Memory behavior: This constructor may allocate objects within the currently active MemoryArea.

@SCJAllowed(javax.safetycritical.annotate.Level.LEVEL_2)
@SCJRestricted({javax.safetycritical.annotate.Phase.ALL},
  maySelfSuspend = true,
  mayAllocate = false)
public static void yield( )

Memory behavior: Allocates no memory. Causes the currently executing thread object to temporary pause and allow other threads to execute.

B.2.46 CLASS Throwable

@SCJAllowed
public class Throwable
  implements java.io.Serializable
  extends java.lang.Object

Constructors

@SCJAllowed
public Throwable( )

Constructs a Throwable object for an SCJ system with a null detail message and no specified cause. Invokes System.captureStackBacktrace(this) to save the back trace associated with the current thread.

This constructor shall not copy this to any instance or static field.

@SCJAllowed
@SCJRestricted({javax.safetycritical.annotate.Phase.ALL},

```
   maySelfSuspend = false,
   mayAllocate = true)
public Throwable(Throwable cause)
```

Constructs a Throwable object for an SCJ system with a null detail message and a specified cause. Does not invoke System.captureStackBacktrace(this) to avoid overwriting the back trace associated with the cause.

This constructor shall not copy this to any instance or static field.

cause — the cause of this exception.

Memory behavior: This constructor requires that the "cause" argument reside in a scope that encloses the scope of the "this" argument.

```
@SCJAllowed
@SCJRestricted({javax.safetycritical.annotate.Phase.ALL},
   maySelfSuspend = false,
   mayAllocate = true)
public Throwable(String msg, Throwable cause)
```

Constructs a Throwable object for an SCJ system with a specified detail message and a specified cause. Does not invoke System.captureStackBacktrace(this) to avoid overwriting the back trace associated with the cause.

This constructor shall not copy this to any instance or static field.

msg — the detail message for this Throwable object.

cause — the cause of this exception.

Memory behavior: This constructor requires that the "cause" argument reside in a scope that encloses the scope of the "this" argument. This constructor requires that the "msg" argument reside in a scope that encloses the scope of the "this" argument.

```
@SCJAllowed
@SCJRestricted({javax.safetycritical.annotate.Phase.ALL},
   maySelfSuspend = false,
   mayAllocate = true)
public Throwable(String msg)
```

Constructs a Throwable object for an SCJ system with a specified detail message and no specified cause. Invokes System.captureStackBacktrace(this) to save the back trace associated with the current thread.

This constructor shall not copy this to any instance or static field.

msg — the detail message for this Throwable object.

Memory behavior: This constructor requires that the "msg" argument reside in a scope that encloses the scope of the "this" argument.

Methods

```
@SCJAllowed
@SCJRestricted({javax.safetycritical.annotate.Phase.ALL},
    maySelfSuspend = false,
    mayAllocate = true)
public java.lang.Throwable getCause( )
```

returns a reference to the same Throwable that was supplied as an argument to the constructor, or null if no cause was specified at construction time. Performs no memory allocation.

```
@SCJAllowed
@SCJRestricted({javax.safetycritical.annotate.Phase.ALL},
    maySelfSuspend = false,
    mayAllocate = true)
public java.lang.String getMessage( )
```

returns a reference to the same String 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
@SCJRestricted({javax.safetycritical.annotate.Phase.ALL},
    maySelfSuspend = false,
    mayAllocate = true)
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 thread-local buffer to represent the stack back trace information associated with the most recent invocation of System.-captureStackBacktrace . The size of this buffer is specified by providing a StorageParameters object as an argument to construction of the Schedulable. Most commonly, System.captureStackBacktrace is invoked from within the constructor of java.lang.Throwable . getStackTrace returns a representation of this thread-local back trace information.

If System.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 back trace for this particular Throwable. Shall not copy this to any instance or static field.

Memory behavior: This constructor may allocate objects within the currently active MemoryArea.

B.2.47 CLASS **UnsatisfiedLinkError**

@SCJAllowed
public class UnsatisfiedLinkError
  implements java.io.Serializable
  extends java.lang.RuntimeException

Constructors

@SCJAllowed @SCJRestricted(maySelfSuspend = false)
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).

Memory behavior: This constructor may allocate objects within the currently active MemoryArea.

@SCJAllowed @SCJRestricted(maySelfSuspend = false)
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 Illegal-AssignmentError will be thrown.

Allocates an application- and implementation-dependent amount of memory in the current scope (to represent stack backtrace).

Memory behavior: This constructor may allocate objects within the currently active MemoryArea.

This constructor requires that the "msg" argument reside in a scope that encloses the scope of the "this" argument.

B.2.48 CLASS **UnsupportedOperationException**

```
@SCJAllowed
public class UnsupportedOperationException
  implements java.io.Serializable
  extends java.lang.RuntimeException
```

Constructors

```
@SCJAllowed
@SCJRestricted({javax.safetycritical.annotate.Phase.ALL},
   maySelfSuspend = false,
   mayAllocate = true)
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).

Memory behavior: This constructor may allocate objects within the currently active MemoryArea.

```
@SCJAllowed
@SCJRestricted({javax.safetycritical.annotate.Phase.ALL},
   maySelfSuspend = false,
   mayAllocate = true)
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).

Memory behavior: This constructor may allocate objects within the currently active MemoryArea.

This constructor requires that the "msg" argument reside in a scope that encloses the scope of the "this" argument.

```
@SCJAllowed
@SCJRestricted({javax.safetycritical.annotate.Phase.ALL},
   maySelfSuspend = false,
   mayAllocate = true)
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).

Memory behavior: This constructor may allocate objects within the currently active MemoryArea.

This constructor requires that the "msg" argument reside in a scope that encloses the scope of the "this" argument. This constructor requires that the "t" argument reside in a scope that encloses the scope of the "this" argument.

```
@SCJAllowed
@SCJRestricted({javax.safetycritical.annotate.Phase.ALL},
   maySelfSuspend = false,
   mayAllocate = true)
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).

Memory behavior: This constructor may allocate objects within the currently active MemoryArea.

This constructor requires that the "t" argument reside in a scope that encloses the scope of the "this" argument.

B.2.49 CLASS **VirtualMachineError**

@SCJAllowed
public class VirtualMachineError
  implements java.io.Serializable
  extends java.lang.Error

Constructors

@SCJAllowed @SCJRestricted(maySelfSuspend = false)
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).

Memory behavior: This constructor may allocate objects within the currently active MemoryArea.

@SCJAllowed @SCJRestricted(maySelfSuspend = false)
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).

Memory behavior: This constructor may allocate objects within the currently active MemoryArea.

This constructor requires that the "msg" argument reside in a scope that encloses the scope of the "this" argument.

B.2.50 CLASS **Void**

@SCJAllowed
public final class Void extends java.lang.Object

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 Interfaces

C.1.1 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
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 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

C.1.2 INTERFACE InputConnection

@SCJAllowed
public interface InputConnection
implements javax.microedition.io.Connection

This interface defines the capabilities that an input stream connection must have.

Methods

@SCJAllowed
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
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.1.3 INTERFACE **OutputConnection**

@SCJAllowed
public interface OutputConnection
implements javax.microedition.io.Connection

This interface defines the capabilities that an output stream connection must have.

Methods

@SCJAllowed
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
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.1.4 INTERFACE **StreamConnection**

@SCJAllowed
public interface StreamConnection
implements 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 (for instance in MIDP), 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.2 Classes

C.2.1 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
public ConnectionNotFoundException( )

Constructs a ConnectionNotFoundException with no detail message.

@SCJAllowed
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.

C.2.2 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 protocolno timeout 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
public static javax.microedition.io.Connection open(String name)
  throws java.io.IOException
```

Create and open a Connection.

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 may be thrown if access to the protocol handler is prohibited.

```
@SCJAllowed
public static javax.microedition.io.Connection open(String name,
  int mode)
  throws java.io.IOException
```

Create and open a Connection.

name — The URL for the connection.

mode — The access mode.

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 may be thrown if access to the protocol handler is prohibited.

@SCJAllowed
public static java.io.DataInputStream openDataInputStream(String name)
  throws java.io.IOException

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 may be thrown if access to the protocol handler is prohibited.

@SCJAllowed
public static java.io.DataOutputStream openDataOutputStream(String name)
  throws java.io.IOException

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 may be thrown if access to the protocol handler is prohibited.

@SCJAllowed
public static java.io.InputStream openInputStream(String name)
  throws java.io.IOException

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 may be thrown if access to the protocol handler is prohibited.

@SCJAllowed
public static java.io.OutputStream openOutputStream(String name)
  throws java.io.IOException

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 may be thrown if access to the protocol handler is prohibited.

Appendix D

Javadoc Description of Package javax.realtime

Package Contents *Page*

D.1 Interfaces

D.1.1 INTERFACE AllocationContext

@SCJAllowed
public interface AllocationContext

All memory allocation takes places from within an allocation context. This interface defines the operations available on all allocation contexts. Allocation contexts are implemented by memory areas. The RTSJenter method is not callable by the SCJ application and hence is omitted.

Methods

@SCJAllowed
public long memoryConsumed( )

returns the amount of memory in bytes consumed so far in this memory area.

@SCJAllowed
public long memoryRemaining( )

returns the amount of memory in bytes remaining in this memory area.

@SCJAllowed @SCJRestricted(maySelfSuspend = false)
public java.lang.Object newArray(**Class**<> type, **int** number)
throws java.lang.IllegalArgumentException

Create an array object of the given type and length number in this memory area.

type — the class of object this memory area should hold. An array of a primitive type can be created using a type such as Integer.TYPE, which would create an array of the int type.

number — the number of elements the array should have.

returns the new array of class type and size number.

Throws IllegalArgumentException if number is less than zero, type is null, or type is java.lang.Void.TYPE.

Throws OutOfMemoryError if space in the memory area is exhausted.

@SCJAllowed
public java.lang.Object newInstance(**Class**<> type)

throws java.lang.IllegalAccessException,
java.lang.InstantiationException,
java.lang.OutOfMemoryError

Create a new instance of class type in this memory area.

type — is the class of the object to be created

returns a new instance of the given class.

Throws IllegalAccessException if the class or constructor is inaccessible due to access rules.

Throws InstantiationException if the specified class object could not be constructed because (1) the class is an interface, (2) the class is abstract, (3) the class is an array, or (4) the class either does not have a no-argument constructor or its no-argument constructor is not visible.

Throws OutOfMemoryError if space in the memory area is exhausted.

@SCJAllowed
public long size( )

returns the current size of this memory area in bytes.

D.1.2 INTERFACE **ClockCallBack**

@SCJAllowed(javax.safetycritical.annotate.Level.LEVEL_1)
public interface ClockCallBack

The ClockEvent interface may be used by subclasses of Clock to indicate to the clock infrastructure that the clock has either reached a designated time, or has experienced a discontinuity. Invocations of the methods in ClockCallBack are serialized.

Methods

@SCJAllowed(javax.safetycritical.annotate.Level.LEVEL_1)
@SCJRestricted(maySelfSuspend = false)
public void atTime(Clock clock)

Clock has reached the designated time. This clock event is de-registered before this method is invoked.

clock — the clock that has reached a designated time.

@SCJAllowed(javax.safetycritical.annotate.Level.LEVEL_1)
@SCJRestricted(maySelfSuspend = false)
public void discontinuity(Clock clock, AbsoluteTime updatedTime)

The clock experienced a time discontinuity (it changed its time value other than by ticking.) The clock has de-registered this clock event.

clock — the clock that has experienced a discontinuity.

updatedTime — the signed length of the time discontinuity.

D.1.3 INTERFACE **EventExaminer**

@SCJAllowed(javax.safetycritical.annotate.Level.LEVEL_2)
public interface EventExaminer

Note:

Methods

@SCJAllowed(javax.safetycritical.annotate.Level.LEVEL_2)
public void visit(AbstractAsyncEvent event)

D.1.4 INTERFACE **Happening**

@SCJAllowed(javax.safetycritical.annotate.Level.LEVEL_1)
public interface Happening

This interface is the common parent type for happenings that are to be used for triggering external events. Happening represent events that can be triggered based on some event external to the VM. This includes POSIX signals and POSIX realtime signals.

Methods

@SCJAllowed(javax.safetycritical.annotate.Level.LEVEL_1)
public void attach(AbstractAsyncEvent event)
  throws java.lang.IllegalArgumentException

Bind an **javax.realtime.AbstractAsyncEvent** to this Happening.

An **javax.realtime.AbstractAsyncEvent** may be bound to more than one signal, and it may be both bound to happenings using AbstractAsyncEvent.bindTo(String) and this method. Also, each signal may be attached to more than one instances of **javax.realtime.AbstractAsyncEvent**

event — to be attached to this signal.

Throws DuplicateEventException when the given event is already attached to the signal

```
@SCJAllowed(javax.safetycritical.annotate.Level.LEVEL_1)
@SCJRestricted(mayAllocate = false,
   maySelfSuspend = false,
   {javax.safetycritical.annotate.Phase.ALL})
```
public void deregister( )

```
@SCJAllowed(javax.safetycritical.annotate.Level.LEVEL_1)
```
public boolean deregisterName( )

Unregister this signal so that it cannot be triggered.

```
@SCJAllowed(javax.safetycritical.annotate.Level.LEVEL_1)
@SCJRestricted(mayAllocate = false,
   maySelfSuspend = false,
   {javax.safetycritical.annotate.Phase.ALL})
```
public int getId( )

Get the ID of this signal.

returns the id of this signal

```
@SCJAllowed(javax.safetycritical.annotate.Level.LEVEL_1)
@SCJRestricted(mayAllocate = false,
   maySelfSuspend = false,
   {javax.safetycritical.annotate.Phase.ALL})
```
public java.lang.String getName( )

Get the name of this signal.

returns the name of this signal.

```
@SCJAllowed(javax.safetycritical.annotate.Level.LEVEL_1)
```
public boolean isHappening(String name)

Determine if a signal with a given name is registered.

name — of the signal

returns true when a signal with the given name is registered

@SCJAllowed(javax.safetycritical.annotate.Level.LEVEL_1)
public boolean isRegistered( )

Indicates if the happening is registered or not.

returns true when the signal is registered

@SCJAllowed(javax.safetycritical.annotate.Level.LEVEL_1)
@SCJRestricted(mayAllocate = false,
  maySelfSuspend = false,
  {javax.safetycritical.annotate.Phase.ALL})
public boolean registerName( )

Register this signal with the given manager so that it can be triggered.

returns true when the happening is registered

@SCJAllowed(javax.safetycritical.annotate.Level.LEVEL_1)
public void unbind(AbstractAsyncEvent event)
  throws javax.realtime.EventNotFoundException

Unbind an **javax.realtime.AbstractAsyncEvent** from this signal.

event — to be unbound from this signal.

Throws EventNotFoundException when the given event is not attached to the given signal.

D.1.5 INTERFACE **PhysicalMemoryName**

@SCJAllowed(javax.safetycritical.annotate.Level.LEVEL_0)
public interface PhysicalMemoryName

D.1.6 INTERFACE **RawByte**

@SCJAllowed(javax.safetycritical.annotate.Level.LEVEL_0)
public interface RawByte
  implements javax.realtime.RawByteRead, javax.realtime.RawByteWrite

An interface to a byte accessor object. An accessor object encapsules the protocol required to access a byte in raw memory.

D.1.7 INTERFACE **RawByteArray**

@SCJAllowed(javax.safetycritical.annotate.Level.LEVEL_0)
public interface RawByteArray
implements javax.realtime.RawByteArrayRead, javax.realtime.RawByteArrayWrite

An interface to a byte array accessor object. An accessor object encapsules the protocol required to access a byte array in raw memory.

D.1.8 INTERFACE **RawByteArrayRead**

@SCJAllowed(javax.safetycritical.annotate.Level.LEVEL_0)
public interface RawByteArrayRead

An interface to a byte array read accessor object. An accessor object encapsules the protocol required to read a byte array in raw memory.

Methods

@SCJAllowed(javax.safetycritical.annotate.Level.LEVEL_0)
@SCJRestricted(mayAllocate = false, maySelfSuspend = false)
public byte get(**long** offset)

Get the value from this raw byte array.

returns the byte from the array in raw memory.

@SCJAllowed(javax.safetycritical.annotate.Level.LEVEL_0)
@SCJRestricted(mayAllocate = false, maySelfSuspend = false)
public void get(**byte** [] array)

Get the value of this raw byte array into array.

D.1.9 INTERFACE **RawByteArrayWrite**

@SCJAllowed(javax.safetycritical.annotate.Level.LEVEL_0)
public interface RawByteArrayWrite

An interface to a byte array write accessor object. An accessor object encapsules the protocol required to write a byte array in raw memory.

Methods

@SCJAllowed(javax.safetycritical.annotate.Level.LEVEL_0)
@SCJRestricted(mayAllocate = false, maySelfSuspend = false)
public void put(**byte** b, **long** offset)

Store the byte in array in the associated Raw memory.

b — is the byte to be stored.

offset — is the location of the byte to be stored relative to the raw memory area.

@SCJAllowed(javax.safetycritical.annotate.Level.LEVEL_0)
@SCJRestricted(mayAllocate = false, maySelfSuspend = false)
public void put(**byte** [] i)

Store the byte array value in the associated Raw memory.

i — is the array to be stored.

D.1.10 INTERFACE **RawByteRead**

@SCJAllowed(javax.safetycritical.annotate.Level.LEVEL_0)
public interface RawByteRead

An interface to a byte accessor object. An accessor object encapsules the protocol required to read a byte in raw memory.

Methods

@SCJAllowed(javax.safetycritical.annotate.Level.LEVEL_0)
@SCJRestricted(mayAllocate = false, maySelfSuspend = false)
public byte get( )

Get the value of this raw byte.

returns the byte from raw memory.

D.1.11 INTERFACE **RawByteWrite**

@SCJAllowed(javax.safetycritical.annotate.Level.LEVEL_0)
public interface RawByteWrite

An interface to a byte accessor object. An accessor object encapsules the protocol required to write a byte in raw memory.

Methods

@SCJAllowed(javax.safetycritical.annotate.Level.LEVEL_0)
@SCJRestricted(mayAllocate = false, maySelfSuspend = false)
public void put(**byte** value)

Store the byte value in the associated raw memory.

value — is the byte to be stored.

D.1.12 INTERFACE **RawInt**

@SCJAllowed(javax.safetycritical.annotate.Level.LEVEL_0)
public interface RawInt
implements javax.realtime.RawIntRead, javax.realtime.RawIntWrite

An interface to a int accessor object. An accessor object encapsulates the protocol required to access a int in raw memory.

D.1.13 INTERFACE **RawIntArray**

@SCJAllowed(javax.safetycritical.annotate.Level.LEVEL_0)
public interface RawIntArray
implements javax.realtime.RawIntArrayRead, javax.realtime.RawIntArrayWrite

An interface to a int array accessor object. An accessor object encapsules the protocol required to access a int array in raw memory.

D.1.14 INTERFACE **RawIntArrayRead**

@SCJAllowed(javax.safetycritical.annotate.Level.LEVEL_0)
public interface RawIntArrayRead

An interface to a int array accessor object. An accessor object encapsules the protocol required to access a int array in raw memory.

Methods

@SCJAllowed(javax.safetycritical.annotate.Level.LEVEL_0)
@SCJRestricted(mayAllocate = false, maySelfSuspend = false)
public int get(**long** offset)

Get the value of a int from this raw int array.

returns the int from raw memory.

@SCJAllowed(javax.safetycritical.annotate.Level.LEVEL_0)
@SCJRestricted(mayAllocate = false, maySelfSuspend = false)
public void get(**int** [] array)

Get the value of this raw int array into array.

array — is the array to place the data.

D.1.15 INTERFACE **RawIntArrayWrite**

@SCJAllowed(javax.safetycritical.annotate.Level.LEVEL_0)
public interface RawIntArrayWrite

An interface to a int array accessor object for writing. An accessor object encapsules the protocol required to write access a int array in raw memory.

Methods

@SCJAllowed(javax.safetycritical.annotate.Level.LEVEL_0)
@SCJRestricted(mayAllocate = false, maySelfSuspend = false)
public void put(**int** value, **long** offset)

Store the int in the associated Raw memory array.

value — is the int to be stored.

offset — is the position in array to be stored.

@SCJAllowed(javax.safetycritical.annotate.Level.LEVEL_0)
@SCJRestricted(mayAllocate = false, maySelfSuspend = false)
public void put(**int** [] array)

Store the int array value in the associated Raw memory.

array — is the array to be stored.

D.1.16 INTERFACE **RawIntRead**

@SCJAllowed(javax.safetycritical.annotate.Level.LEVEL_0)
public interface RawIntRead

An interface to a int accessor object. An accessor object encapsulates the protocol required to read a int in raw memory.

Methods

@SCJAllowed(javax.safetycritical.annotate.Level.LEVEL_0)
@SCJRestricted(mayAllocate = false, maySelfSuspend = false)
public int get( )

Get the value of this raw int.

returns the int from raw memory.

D.1.17 INTERFACE **RawIntWrite**

@SCJAllowed(javax.safetycritical.annotate.Level.LEVEL_0)
public interface RawIntWrite

An interface to a int accessor object. An accessor object encapsulates the protocol required to write a int in raw memory.

Methods

@SCJAllowed(javax.safetycritical.annotate.Level.LEVEL_0)
@SCJRestricted(mayAllocate = false, maySelfSuspend = false)
public void put(**int** value)

Store the int value in the associated Raw memory.

value — is the value to be stored.

D.1.18 INTERFACE **RawIntegralAccessFactory**

@SCJAllowed(javax.safetycritical.annotate.Level.LEVEL_0)
public interface RawIntegralAccessFactory

An interface that describes factory classes that create the accessor objects for raw memory access.

Methods

@SCJAllowed(javax.safetycritical.annotate.Level.LEVEL_0)
@SCJRestricted(mayAllocate = false, maySelfSuspend = false)
public javax.realtime.RawMemoryName getName( )

returns a reference to an object that implements the RawMemoryName interface. This "name" is associated with this factory and indirectly with all the objects created by this factory.

@SCJAllowed(javax.safetycritical.annotate.Level.LEVEL_0)
@SCJRestricted({javax.safetycritical.annotate.Phase.
  INTERRUPT_SERVICE_ROUTINE}, mayAllocate = false, maySelfSuspend = false)
public javax.realtime.RawByte newRawByte(**long** offset)

Creates an accessor object for accessing a byte in raw memory.

returns an object implementing the RawByte interface.

Throws AlignmentError if the offset is not on the appropriate boundary.

Throws SizeOutOfBoundsException if the byte falls in an invalid address range.

Throws MemoryTypeConflictException if offset does not point to memory that matches the type served by this factory.

Throws OffsetOutOfBoundsException if the offset is negative or greater than the size of the raw memory area.

@SCJAllowed(javax.safetycritical.annotate.Level.LEVEL_0)
@SCJRestricted({javax.safetycritical.annotate.Phase.
  INTERRUPT_SERVICE_ROUTINE}, mayAllocate = false, maySelfSuspend = false)
public javax.realtime.RawByteArray newRawByteArray(**long** base,
 int entries)

Creates an accessor object for accessing a byte array in raw memory.

returns an object implementing the RawByteArray interface.

Throws AlignmentError if the base is not on the appropriate boundary.

Throws SizeOutOfBoundsException if the byte array falls in an invalid address range.

Throws MemoryTypeConflictException if base does not point to memory that matches the type served by this factory.

Throws OffsetOutOfBoundsException if the base is negative or greater than the size of the raw memory area.

```
@SCJAllowed(javax.safetycritical.annotate.Level.LEVEL_0)
@SCJRestricted({javax.safetycritical.annotate.Phase.
   INTERRUPT_SERVICE_ROUTINE}, mayAllocate = false, maySelfSuspend = false)
public javax.realtime.RawByteArrayRead newRawByteArrayRead(long base,
  int entries)
```

Creates an accessor object for read accessing a byte array in raw memory.

returns an object implementing the RawByteArray interface.

Throws AlignmentError if the base is not on the appropriate boundary.

Throws SizeOutOfBoundsException if the byte array falls in an invalid address range.

Throws MemoryTypeConflictException if base does not point to memory that matches the type served by this factory.

Throws OffsetOutOfBoundsException if the base is negative or greater than the size of the raw memory area.

```
@SCJAllowed(javax.safetycritical.annotate.Level.LEVEL_0)
@SCJRestricted({javax.safetycritical.annotate.Phase.
   INTERRUPT_SERVICE_ROUTINE}, mayAllocate = false, maySelfSuspend = false)
public javax.realtime.RawByteArrayWrite newRawByteArrayWrite(
  long base,
  int entries)
```

Creates an accessor object for write accessing a byte array in raw memory.

returns an object implementing the RawByteArray interface.

Throws AlignmentError if the base is not on the appropriate boundary.

Throws SizeOutOfBoundsException if the byte array falls in an invalid address range.

Throws MemoryTypeConflictException if base does not point to memory that matches the type served by this factory.

Throws OffsetOutOfBoundsException if the base is negative or greater than the size of the raw memory area.

```
@SCJAllowed(javax.safetycritical.annotate.Level.LEVEL_0)
@SCJRestricted({javax.safetycritical.annotate.Phase.
   INTERRUPT_SERVICE_ROUTINE}, mayAllocate = false, maySelfSuspend = false)
public javax.realtime.RawByteRead newRawByteRead(long offset)
```

Creates an accessor object for read accessing a byte in raw memory.

returns an object implementing the RawByte interface.

Throws AlignmentError if the offset is not on the appropriate boundary.

Throws SizeOutOfBoundsException if the byte falls in an invalid address range.

Throws MemoryTypeConflictException if offset does not point to memory that matches the type served by this factory.

Throws OffsetOutOfBoundsException if the offset is negative or greater than the size of the raw memory area.

```
@SCJAllowed(javax.safetycritical.annotate.Level.LEVEL_0)
@SCJRestricted({javax.safetycritical.annotate.Phase.
   INTERRUPT_SERVICE_ROUTINE}, mayAllocate = false, maySelfSuspend = false)
public javax.realtime.RawByteWrite newRawByteWrite(long offset)
```

Creates an accessor object for write accessing a byte in raw memory.

returns an object implementing the RawByte interface.

Throws AlignmentError if the offset is not on the appropriate boundary.

Throws SizeOutOfBoundsException if the byte falls in an invalid address range.

Throws MemoryTypeConflictException if offset does not point to memory that matches the type served by this factory.

Throws OffsetOutOfBoundsException if the offset is negative or greater than the size of the raw memory area.

```
@SCJAllowed(javax.safetycritical.annotate.Level.LEVEL_0)
@SCJRestricted({javax.safetycritical.annotate.Phase.
   INTERRUPT_SERVICE_ROUTINE}, mayAllocate = false, maySelfSuspend = false)
public javax.realtime.RawInt newRawInt(long offset)
```

Creates an accessor object for accessing an int in raw memory.

returns an object implementing the RawInt interface.

Throws AlignmentError if the offset is not on the appropriate boundary.

Throws SizeOutOfBoundsException if the int falls in an invalid address range.

Throws OffsetOutOfBoundsException if the offset is negative or greater than the size of the raw memory area.

Throws MemoryTypeConflictException if offset does not point to memory that matches the type served by this factory.

```
@SCJAllowed(javax.safetycritical.annotate.Level.LEVEL_0)
@SCJRestricted({javax.safetycritical.annotate.Phase.
   INTERRUPT_SERVICE_ROUTINE}, mayAllocate = false, maySelfSuspend = false)
public javax.realtime.RawIntArray newRawIntArray(long base, int entries)
```

Creates an accessor object for accessing a int array in raw memory.

returns an object implementing the RawIntArray interface.

Throws AlignmentError if the base is not on the appropriate boundary.

Throws SizeOutOfBoundsException if the int array falls in an invalid address range.

Throws MemoryTypeConflictException if base does not point to memory that matches the type served by this factory.

Throws OffsetOutOfBoundsException if the base is negative or greater than the size of the raw memory area.

```
@SCJAllowed(javax.safetycritical.annotate.Level.LEVEL_0)
@SCJRestricted({javax.safetycritical.annotate.Phase.
   INTERRUPT_SERVICE_ROUTINE}, mayAllocate = false, maySelfSuspend = false)
public javax.realtime.RawIntArrayRead newRawIntArrayRead(long base,
  int entries)
```

Creates an accessor object for read accessing a int array in raw memory.

returns an object implementing the RawIntArray interface.

Throws AlignmentError if the base is not on the appropriate boundary.

Throws SizeOutOfBoundsException if the int array falls in an invalid address range.

Throws MemoryTypeConflictException if base does not point to memory that matches the type served by this factory.

Throws OffsetOutOfBoundsException if the base is negative or greater than the size of the raw memory area.

```
@SCJAllowed(javax.safetycritical.annotate.Level.LEVEL_0)
@SCJRestricted({javax.safetycritical.annotate.Phase.
   INTERRUPT_SERVICE_ROUTINE}, mayAllocate = false, maySelfSuspend = false)
public javax.realtime.RawIntArrayWrite newRawIntArrayWrite(long base,
  int entries)
```

Creates an accessor object for write accessing a int array in raw memory.

returns an object implementing the RawIntArray interface.

Throws AlignmentError if the base is not on the appropriate boundary.

Throws SizeOutOfBoundsException if the int array falls in an invalid address range.

Throws MemoryTypeConflictException if base does not point to memory that matches the type served by this factory.

Throws OffsetOutOfBoundsException if the base is negative or greater than the size of the raw memory area.

```
@SCJAllowed(javax.safetycritical.annotate.Level.LEVEL_0)
@SCJRestricted({javax.safetycritical.annotate.Phase.
   INTERRUPT_SERVICE_ROUTINE}, mayAllocate = false, maySelfSuspend = false)
public javax.realtime.RawIntRead newRawIntRead(long offset)
```

Creates an accessor object for read accessing an int in raw memory.

returns an object implementing the RawInt interface.

Throws AlignmentError if the offset is not on the appropriate boundary.

Throws SizeOutOfBoundsException if the int falls in an invalid address range.

Throws OffsetOutOfBoundsException if the offset is negative or greater than the size of the raw memory area.

Throws MemoryTypeConflictException if offset does not point to memory that matches the type served by this factory.

```
@SCJAllowed(javax.safetycritical.annotate.Level.LEVEL_0)
@SCJRestricted({javax.safetycritical.annotate.Phase.
   INTERRUPT_SERVICE_ROUTINE}, mayAllocate = false, maySelfSuspend = false)
public javax.realtime.RawIntWrite newRawIntWrite(long offset)
```

Creates an accessor object for write accessing an int in raw memory.

returns an object implementing the RawInt interface.

Throws AlignmentError if the offset is not on the appropriate boundary.

Throws SizeOutOfBoundsException if the int falls in an invalid address range.

Throws OffsetOutOfBoundsException if the offset is negative or greater than the size of the raw memory area.

Throws MemoryTypeConflictException if offset does not point to memory that matches the type served by this factory.

```
@SCJAllowed(javax.safetycritical.annotate.Level.LEVEL_0)
@SCJRestricted({javax.safetycritical.annotate.Phase.
   INTERRUPT_SERVICE_ROUTINE}, mayAllocate = false, maySelfSuspend = false)
public javax.realtime.RawLong newRawLong(long offset)
```

Creates an accessor object for accessing a long in raw memory.

returns an object implementing the RawLong interface.

Throws AlignmentError if the offset is not on the appropriate boundary.

Throws SizeOutOfBoundsException if the long falls in an invalid address range.

Throws OffsetOutOfBoundsException if the offset is negative or greater than the size of the raw memory area.

Throws MemoryTypeConflictException if offset does not point to memory that matches the type served by this factory.

```
@SCJAllowed(javax.safetycritical.annotate.Level.LEVEL_0)
@SCJRestricted({javax.safetycritical.annotate.Phase.
   INTERRUPT_SERVICE_ROUTINE}, mayAllocate = false, maySelfSuspend = false)
public javax.realtime.RawLongArray newRawLongArray(long base,
  int entries)
```

Creates an accessor object for accessing a long array in raw memory.

returns an object implementing the RawLongArray interface.

Throws AlignmentError if the base is not on the appropriate boundary.

Throws SizeOutOfBoundsException if the long array falls in an invalid address range.

Throws MemoryTypeConflictException if base does not point to memory that matches the type served by this factory.

Throws OffsetOutOfBoundsException if the base is negative or greater than the size of the raw memory area.

```
@SCJAllowed(javax.safetycritical.annotate.Level.LEVEL_0)
@SCJRestricted({javax.safetycritical.annotate.Phase.
   INTERRUPT_SERVICE_ROUTINE}, mayAllocate = false, maySelfSuspend = false)
public javax.realtime.RawLongArrayRead newRawLongArrayRead(long base,
  int entries)
```

Creates an accessor object for read accessing a long array in raw memory.

returns an object implementing the RawLongArray interface.

Throws AlignmentError if the base is not on the appropriate boundary.

Throws SizeOutOfBoundsException if the long array falls in an invalid address range.

Throws MemoryTypeConflictException if base does not point to memory that matches the type served by this factory.

Throws OffsetOutOfBoundsException if the base is negative or greater than the size of the raw memory area.

```
@SCJAllowed(javax.safetycritical.annotate.Level.LEVEL_0)
@SCJRestricted({javax.safetycritical.annotate.Phase.
   INTERRUPT_SERVICE_ROUTINE}, mayAllocate = false, maySelfSuspend = false)
public javax.realtime.RawLongArrayWrite newRawLongArrayWrite(
  long base,
  int entries)
```

Creates an accessor object for write accessing a long array in raw memory.

returns an object implementing the RawLongArray interface.

Throws AlignmentError if the base is not on the appropriate boundary.

Throws SizeOutOfBoundsException if the long array falls in an invalid address range.

Throws MemoryTypeConflictException if base does not point to memory that matches the type served by this factory.

Throws OffsetOutOfBoundsException if the base is negative or greater than the size of the raw memory area.

```
@SCJAllowed(javax.safetycritical.annotate.Level.LEVEL_0)
@SCJRestricted({javax.safetycritical.annotate.Phase.
   INTERRUPT_SERVICE_ROUTINE}, mayAllocate = false, maySelfSuspend = false)
public javax.realtime.RawLongRead newRawLongRead(long offset)
```

Creates an accessor object for read accessing a long in raw memory.

returns an object implementing the RawLong interface.

Throws AlignmentError if the offset is not on the appropriate boundary.

Throws SizeOutOfBoundsException if the long falls in an invalid address range.

Throws OffsetOutOfBoundsException if the offset is negative or greater than the size of the raw memory area.

Throws MemoryTypeConflictException if offset does not point to memory that matches the type served by this factory.

```
@SCJAllowed(javax.safetycritical.annotate.Level.LEVEL_0)
@SCJRestricted({javax.safetycritical.annotate.Phase.
   INTERRUPT_SERVICE_ROUTINE}, mayAllocate = false, maySelfSuspend = false)
public javax.realtime.RawLongWrite newRawLongWrite(long offset)
```

Creates an accessor object for write accessing a long in raw memory.

returns an object implementing the RawLong interface.

Throws AlignmentError if the offset is not on the appropriate boundary.

Throws SizeOutOfBoundsException if the long falls in an invalid address range.

Throws OffsetOutOfBoundsException if the offset is negative or greater than the size of the raw memory area.

Throws MemoryTypeConflictException if offset does not point to memory that matches the type served by this factory.

```
@SCJAllowed(javax.safetycritical.annotate.Level.LEVEL_0)
@SCJRestricted({javax.safetycritical.annotate.Phase.
   INTERRUPT_SERVICE_ROUTINE}, mayAllocate = false, maySelfSuspend = false)
public javax.realtime.RawShort newRawShort(long offset)
```

Creates an accessor object for accessing a short in raw memory.

Throws AlignmentError if the offset is not on the appropriate boundary.

returns an object implementing the RawShort interface.

Throws SizeOutOfBoundsException if the short falls in an invalid address range.

Throws OffsetOutOfBoundsException if the offset is negative or greater than the size of the raw memory area.

Throws MemoryTypeConflictException if offset does not point to memory that matches the type served by this factory.

```
@SCJAllowed(javax.safetycritical.annotate.Level.LEVEL_0)
@SCJRestricted({javax.safetycritical.annotate.Phase.
   INTERRUPT_SERVICE_ROUTINE}, mayAllocate = false, maySelfSuspend = false)
public javax.realtime.RawShortArray newRawShortArray(long base,
  int entries)
```

Creates an accessor object for accessing a short array in raw memory.

returns an object implementing the RawShortArray interface.

Throws AlignmentError if the base is not on the appropriate boundary.

Throws SizeOutOfBoundsException if the short array falls in an invalid address range.

Throws MemoryTypeConflictException if base does not point to memory that matches the type served by this factory.

Throws OffsetOutOfBoundsException if the base is negative or greater than the size of the raw memory area.

```
@SCJAllowed(javax.safetycritical.annotate.Level.LEVEL_0)
@SCJRestricted({javax.safetycritical.annotate.Phase.
   INTERRUPT_SERVICE_ROUTINE}, mayAllocate = false, maySelfSuspend = false)
public javax.realtime.RawShortArrayRead newRawShortArrayRead(
  long base,
  int entries)
```

Creates an accessor object for read accessing a short array in raw memory.

returns an object implementing the RawShortArray interface.

Throws AlignmentError if the base is not on the appropriate boundary.

Throws SizeOutOfBoundsException if the short array falls in an invalid address range.

Throws MemoryTypeConflictException if base does not point to memory that matches the type served by this factory.

Throws OffsetOutOfBoundsException if the base is negative or greater than the size of the raw memory area.

```
@SCJAllowed(javax.safetycritical.annotate.Level.LEVEL_0)
@SCJRestricted({javax.safetycritical.annotate.Phase.
   INTERRUPT_SERVICE_ROUTINE}, mayAllocate = false, maySelfSuspend = false)
public javax.realtime.RawShortArrayWrite newRawShortArrayWrite(
  long base,
  int entries)
```

Creates an accessor object for write accessing a short array in raw memory.

returns an object implementing the RawShortArray interface.

Throws AlignmentError if the base is not on the appropriate boundary.

Throws SizeOutOfBoundsException if the short array falls in an invalid address range.

Throws MemoryTypeConflictException if base does not point to memory that matches the type served by this factory.

Throws OffsetOutOfBoundsException if the base is negative or greater than the size of the raw memory area.

```
@SCJAllowed(javax.safetycritical.annotate.Level.LEVEL_0)
@SCJRestricted({javax.safetycritical.annotate.Phase.
   INTERRUPT_SERVICE_ROUTINE}, mayAllocate = false, maySelfSuspend = false)
public javax.realtime.RawShortRead newRawShortRead(long offset)
```

Creates an accessor object for read accessing a short in raw memory.

Throws AlignmentError if the offset is not on the appropriate boundary.

returns an object implementing the RawShort interface.

Throws SizeOutOfBoundsException if the short falls in an invalid address range.

Throws OffsetOutOfBoundsException if the offset is negative or greater than the size of the raw memory area.

Throws MemoryTypeConflictException if offset does not point to memory that matches the type served by this factory.

```
@SCJAllowed(javax.safetycritical.annotate.Level.LEVEL_0)
@SCJRestricted({javax.safetycritical.annotate.Phase.
   INTERRUPT_SERVICE_ROUTINE}, mayAllocate = false, maySelfSuspend = false)
public javax.realtime.RawShortWrite newRawShortWrite(long offset)
```

Creates an accessor object for write accessing a short in raw memory.

returns an object implementing the RawShort interface.

Throws AlignmentError if the offset is not on the appropriate boundary.

Throws SizeOutOfBoundsException if the short falls in an invalid address range.

Throws OffsetOutOfBoundsException if the offset is negative or greater than the size of the raw memory area.

Throws MemoryTypeConflictException if offset does not point to memory that matches the type served by this factory.

D.1.19 INTERFACE **RawLong**

```
@SCJAllowed(javax.safetycritical.annotate.Level.LEVEL_0)
public interface RawLong
  implements javax.realtime.RawLongRead, javax.realtime.RawLongWrite
```

An interface to a long accessor object. An accessor object encapsulates the protocol required to access a long in raw memory.

D.1.20 INTERFACE **RawLongArray**

@SCJAllowed(javax.safetycritical.annotate.Level.LEVEL_0)
public interface RawLongArray
implements javax.realtime.RawLongArrayRead, javax.realtime.RawLongArrayWrite

An interface to a long array accessor object. An accessor object encapsules the protocol required to access a long array in raw memory.

D.1.21 INTERFACE **RawLongArrayRead**

@SCJAllowed(javax.safetycritical.annotate.Level.LEVEL_0)
public interface RawLongArrayRead

An longerface to a long array accessor object. An accessor object encapsules the protocol required to access a long array in raw memory.

Methods

@SCJAllowed(javax.safetycritical.annotate.Level.LEVEL_0)
@SCJRestricted(mayAllocate = false, maySelfSuspend = false)
public long get(**long** offset)

Get the value of a long from this raw long array.

returns the long from raw memory.

@SCJAllowed(javax.safetycritical.annotate.Level.LEVEL_0)
@SCJRestricted(mayAllocate = false, maySelfSuspend = false)
public void get(**long** [] array)

Get the value of this raw long array longo array.

array — is the array to place the data.

D.1.22 INTERFACE **RawLongArrayWrite**

@SCJAllowed(javax.safetycritical.annotate.Level.LEVEL_0)
public interface RawLongArrayWrite

An longerface to a long array accessor object for writing. An accessor object encapsules the protocol required to write access a long array in raw memory.

Methods

@SCJAllowed(javax.safetycritical.annotate.Level.LEVEL_0)
@SCJRestricted(mayAllocate = false, maySelfSuspend = false)
public void put(**long** value, **long** offset)

Store the long in the associated Raw memory array.

value — is the long to be stored.

offset — is the position in array to be stored.

@SCJAllowed(javax.safetycritical.annotate.Level.LEVEL_0)
@SCJRestricted(mayAllocate = false, maySelfSuspend = false)
public void put(**long** [] array)

Store the long array value in the associated Raw memory.

array — is the array to be stored.

D.1.23 INTERFACE **RawLongRead**

@SCJAllowed(javax.safetycritical.annotate.Level.LEVEL_0)
public interface RawLongRead

An interface to a long accessor object. An accessor object encapsulates the protocol required to read a long in raw memory.

Methods

@SCJAllowed(javax.safetycritical.annotate.Level.LEVEL_0)
@SCJRestricted(mayAllocate = false, maySelfSuspend = false)
public long get( )

Get the value of this raw long.

returns the long from raw memory.

D.1.24 INTERFACE **RawLongWrite**

@SCJAllowed(javax.safetycritical.annotate.Level.LEVEL_0)
public interface RawLongWrite

An interface to a long accessor object. An accessor object encapsulates the protocol required to write a long in raw memory.

Methods

@SCJAllowed(javax.safetycritical.annotate.Level.LEVEL_0)
@SCJRestricted(mayAllocate = false, maySelfSuspend = false)
public void put(**long** value)

Store the long value in the associated Raw memory.

value — is the value to be stored.

D.1.25 INTERFACE **RawMemoryName**

@SCJAllowed(javax.safetycritical.annotate.Level.LEVEL_0)
public interface RawMemoryName

A tagging interface for marking objects that identify raw memory types.

D.1.26 INTERFACE **RawShort**

@SCJAllowed(javax.safetycritical.annotate.Level.LEVEL_0)
public interface RawShort
implements javax.realtime.RawShortRead, javax.realtime.RawShortWrite

An interface to a short accessor object. An accessor object encapsulates the protocol required to access a short in raw memory.

D.1.27 INTERFACE **RawShortArray**

@SCJAllowed(javax.safetycritical.annotate.Level.LEVEL_0)
public interface RawShortArray
implements javax.realtime.RawShortArrayRead, javax.realtime.RawShortArrayWrite

An interface to a short array accessor object. An accessor object encapsules the protocol required to access a short array in raw memory.

D.1.28 INTERFACE **RawShortArrayRead**

@SCJAllowed(javax.safetycritical.annotate.Level.LEVEL_0)
public interface RawShortArrayRead

An interface to a short array accessor object. An accessor object encapsules the protocol required to access a short array in raw memory.

Methods

@SCJAllowed(javax.safetycritical.annotate.Level.LEVEL_0)
@SCJRestricted(mayAllocate = false,
  maySelfSuspend = false,
  {javax.safetycritical.annotate.Phase.ALL})
public short get(**long** offset)

Get the value of a short from this raw short array.

returns the short from raw memory.

@SCJAllowed(javax.safetycritical.annotate.Level.LEVEL_0)
@SCJRestricted(mayAllocate = false,
  maySelfSuspend = false,
  {javax.safetycritical.annotate.Phase.ALL})
public void get(**short** [] array)

Get the value of this raw short array into array.

array — is the array to place the data.

D.1.29 INTERFACE **RawShortArrayWrite**

@SCJAllowed(javax.safetycritical.annotate.Level.LEVEL_0)
public interface RawShortArrayWrite

An interface to a short array accessor object for writing. An accessor object encapsules the protocol required to write access a short array in raw memory.

Methods

@SCJAllowed(javax.safetycritical.annotate.Level.LEVEL_0)
@SCJRestricted(mayAllocate = false,
  maySelfSuspend = false,
  {javax.safetycritical.annotate.Phase.ALL})
public void put(**short** value, **long** offset)

Store the short in the associated Raw memory array.

value — is the short to be stored.

offset — is the position in array to be stored.

```
@SCJAllowed(javax.safetycritical.annotate.Level.LEVEL_0)
@SCJRestricted(mayAllocate = false,
   maySelfSuspend = false,
   {javax.safetycritical.annotate.Phase.ALL})
public void put(short [] array)
```

Store the short array value in the associated Raw memory.

array — is the array to be stored.

D.1.30 INTERFACE **RawShortRead**

```
@SCJAllowed(javax.safetycritical.annotate.Level.LEVEL_0)
public interface RawShortRead
```

An interface to a short accessor object. An accessor object encapsulates the protocol required to read a short in raw memory.

Methods

```
@SCJAllowed(javax.safetycritical.annotate.Level.LEVEL_0)
@SCJRestricted(mayAllocate = false,
   maySelfSuspend = false,
   {javax.safetycritical.annotate.Phase.ALL})
public short get( )
```

Get the value of this raw short.

returns the short from raw memory.

D.1.31 INTERFACE **RawShortWrite**

```
@SCJAllowed(javax.safetycritical.annotate.Level.LEVEL_0)
public interface RawShortWrite
```

An interface to a short accessor object. An accessor object encapsulates the protocol required to write a short in raw memory.

Methods

@SCJAllowed(javax.safetycritical.annotate.Level.LEVEL_0)
@SCJRestricted(mayAllocate = false,
  maySelfSuspend = false,
  {javax.safetycritical.annotate.Phase.ALL})
public void set(**short** value)

Store the short value in the associated Raw memory.

value — is the short to be stored.

D.1.32 INTERFACE **Schedulable**

@SCJAllowed
public interface Schedulable **implements** java.lang.Runnable

In keeping with the RTSJ, SCJ event handlers are schedulable objects. However, the Schedulable interface in the RTSJ is mainly concerned with on-line feasibility analysis and the getting and setting of the parameter classes. On the contrary, in SCJ, it provides no extra functionality over the Runnable interface.

D.1.33 INTERFACE **ScopedAllocationContext**

@SCJAllowed
public interface ScopedAllocationContext
 implements javax.realtime.AllocationContext

This is the base interface for all scoped memory areas. Scoped memory is a region based memory management strategy that can only be cleared when no thread is executing in the area.

D.2 Classes

D.2.1 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 the 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. 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.

Constructors

```
@SCJAllowed
@SCJRestricted({javax.safetycritical.annotate.Phase.ALL},
   maySelfSuspend = false,
   mayAllocate = true)
public AbsoluteTime(long millis, int nanos)
```

Construct an AbsoluteTime object with time millisecond and nanosecond components past the real-time clock's Epoch.

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.

```
@SCJAllowed
@SCJRestricted({javax.safetycritical.annotate.Phase.ALL},
   maySelfSuspend = false,
   mayAllocate = true)
public AbsoluteTime( )
```

Equivalent to new AbsoluteTime(0,0).

```
@SCJAllowed
@SCJRestricted({javax.safetycritical.annotate.Phase.ALL},
   maySelfSuspend = false,
   mayAllocate = true)
public AbsoluteTime(AbsoluteTime time)
```

Make a new AbsoluteTime object from the given AbsoluteTime object.

time — AbsoluteTime object which is the source for the copy.

Memory behavior: This constructor requires that the "time.getClock()" argument reside in a scope that encloses the scope of the "this" argument.

```
@SCJAllowed
@SCJRestricted({javax.safetycritical.annotate.Phase.ALL},
   maySelfSuspend = false,
   mayAllocate = true)
public AbsoluteTime(long millis, int nanos, Clock clock)
```

Construct an AbsoluteTime object with time millisecond and nanosecond components past the epoch for clock.

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 clock providing the association for the newly constructed object.

```
@SCJAllowed
@SCJRestricted({javax.safetycritical.annotate.Phase.ALL},
   maySelfSuspend = false,
   mayAllocate = true)
public AbsoluteTime(Clock clock)
```

Equivalent to new AbsoluteTime(0,0,clock).

clock — The clock providing the association for the newly constructed object.

Methods

```
@SCJAllowed
@SCJRestricted({javax.safetycritical.annotate.Phase.ALL},
   maySelfSuspend = false,
   mayAllocate = true)
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.

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 result of the normalization of this plus millis and nanos in dest if dest is not null, otherwise the result is returned in a newly allocated object.

```
@SCJAllowed
@SCJRestricted({javax.safetycritical.annotate.Phase.ALL},
   maySelfSuspend = false,
   mayAllocate = true)
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.

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 result of the normalization of this plus the RelativeTime parameter time in dest if dest is not null, otherwise the result is returned in a newly allocated object.

```
@SCJAllowed
@SCJRestricted({javax.safetycritical.annotate.Phase.ALL},
   maySelfSuspend = false,
   mayAllocate = true)
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.

time — The time to add to this.

returns A new AbsoluteTime object whose time is the normalization of this plus the parameter time.

Memory behavior: This constructor may allocate objects within the currently active MemoryArea.

```
@SCJAllowed
@SCJRestricted({javax.safetycritical.annotate.Phase.ALL},
    maySelfSuspend = false,
    mayAllocate = true)
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.

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.

Memory behavior: This constructor may allocate objects within the currently active MemoryArea.

```
@SCJAllowed
@SCJRestricted({javax.safetycritical.annotate.Phase.ALL},
    maySelfSuspend = false,
    mayAllocate = true)
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.

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 result of the normalization of this minus the AbsoluteTime parameter time in dest if dest is not null, otherwise the result is returned in a newly allocated object.

```
@SCJAllowed
@SCJRestricted({javax.safetycritical.annotate.Phase.ALL},
    maySelfSuspend = false,
    mayAllocate = true)
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 result of the normalization of this minus the RelativeTime parameter time in dest if dest is not null, otherwise the result is returned in a newly allocated object.

```
@SCJAllowed
@SCJRestricted({javax.safetycritical.annotate.Phase.ALL},
   maySelfSuspend = false,
   mayAllocate = true)
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.

time — The time to subtract from this.

returns A new AbsoluteTime object whose time is the normalization of this minus the parameter time.

Memory behavior: This constructor may allocate objects within the currently active MemoryArea.

```
@SCJAllowed
@SCJRestricted({javax.safetycritical.annotate.Phase.ALL},
   maySelfSuspend = false,
   mayAllocate = true)
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.

time — The time to subtract from this.

returns A new RelativeTime object whose time is the normalization of this minus the AbsoluteTime parameter time.

Memory behavior: This constructor may allocate objects within the currently active MemoryArea.

D.2.2 CLASS **AbstractAsyncEvent**

@SCJAllowed(javax.safetycritical.annotate.Level.LEVEL_1)
public abstract class AbstractAsyncEvent **extends** java.lang.Object

This is the base class for all asynchronous events. Handlers are attached to events by the infrastructure. Consequently the related methods are not visible and the class is empty.

D.2.3 CLASS **AbstractAsyncEventHandler**

@SCJAllowed(javax.safetycritical.annotate.Level.LEVEL_0)
public abstract class AbstractAsyncEventHandler
  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.

D.2.4 CLASS **AffinitySet**

@SCJAllowed
public final class AffinitySet **extends** java.lang.Object

This class is the API for all processor-affinity-related aspects of SCJ. It includes a factory that generates AffinitySet objects, and methods that control the default affinity sets used when affinity set inheritance does not apply.

Affinity sets implement the concept of SCJ scheduling allocation domains. They provide the mechanism by which the programmer can specify the processors on which managed schedulable objects can execute.

The processor membership of an affinity set is immutable. SCJ constrains the use of RTSJ affinity sets so that the affinity of a managed schedulable object can only be set during the initialization phase.

The internal representation of a set of processors in an affinity set instance is not specified. Each processor/core in the system is given a unique logical number.

The relationship between logical and physical processors is implementation-defined.

The affinity set factory cannot create an affinity set with more than one processor member, but such affinity sets are supported as pre-defined affinity sets at Level 2.

A managed schedulable object inherits its creator's affinity set. Every managed schedulable object is associated with a processor affinity set instance, either explicitly assigned, inherited, or defaulted.

See also Services.getSchedulingAllocationDoamins()

Methods

@SCJAllowed(javax.safetycritical.annotate.Level.LEVEL_2)
public static javax.realtime.AffinitySet generate(**int** processorNumber)

Generates an affinity set consisting of a single processor.

returns An AffinitySet representing a single processors in the system. The returned object may be dynamically created in the current memory area or preallocated in immortal memory.

Throws IllegalArgumentException if processorNumber is not a valid processor in the set of processors allocated to the JVM.

@SCJAllowed(javax.safetycritical.annotate.Level.LEVEL_2)
public static final javax.realtime.AffinitySet getAffinitySet(
  Thread thread)

returns an AffinitySet representing the set of processors on which thread can be scheduled. The returned object may be dynamically created in the current memory area or preallocated in immortal memory.

Throws throw NullPointerException if thread is null.

@SCJAllowed(javax.safetycritical.annotate.Level.LEVEL_1)
public static final javax.realtime.AffinitySet getAffinitySet(
  BoundAsyncEventHandler handler)

returns an AffinitySet representing the set of processors on which handler can be scheduled. The returned object may be dynamically created in the current memory area or preallocated in immortal memory.

Throws NullPointerException if handler is null.

@SCJAllowed(javax.safetycritical.annotate.Level.LEVEL_2)
public final boolean isProcessorInSet(**int** processorNumber)

true if and only if the processorNumber is in this affinity set.

@SCJAllowed(javax.safetycritical.annotate.Level.LEVEL_2)
public static final void setProcessorAffinity(AffinitySet set,
Thread thread)

Set the set of processors on which thread can be scheduled to that represented by set}.

set — is the required affinity set

thread — is the target managed thread.

Throws ProcessorAffinityException if set is not a valid processor set, and NullPointerException if thread is null

@SCJAllowed(javax.safetycritical.annotate.Level.LEVEL_1)
public static final void setProcessorAffinity(AffinitySet set,
BoundAsyncEventHandler aeh)

Set the set of processors on which aeh can be scheduled to that represented by set.

set — is the required affinity set

aeh — is the taget bound async event handler

Throws ProcessorAffinityException if set is not a valid processor set, and NullPointerException if handler is null

D.2.5 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)
public AperiodicParameters( )

Construct a new AperiodicParameters object within the current memory area with no deadline detection facility.

@SCJAllowed(javax.safetycritical.annotate.Level.LEVEL_1)
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.

D.2.6 CLASS **AsyncEvent**

@SCJAllowed(javax.safetycritical.annotate.Level.LEVEL_1)
public class AsyncEvent **extends** javax.realtime.AbstractAsyncEvent

In SCJ only aperiodic events are visible at the application level, and they are created only by the \scj infrastructure. Hence, constructors are hidden from public view. Handlers are attached to events by the infrastructure. Consequently the related methods are not visible. There is also no support for binding to external happenings using this class.

Methods

@SCJAllowed(javax.safetycritical.annotate.Level.LEVEL_1)
public void fire( )

Fire this event, i.e., release all of the handlers that were previously associated with this event.

Memory behavior: Does not allocate memory. Does not allow this to escape local variables.

D.2.7 CLASS **AsyncEventHandler**

@SCJAllowed(javax.safetycritical.annotate.Level.LEVEL_0)
public class AsyncEventHandler

extends javax.realtime.AbstractAsyncEventHandler

In SCJ, all asynchronous events must 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)
public void handleAsyncEvent( )

This method must be overridden by the application to provide the handling code.

D.2.8 CLASS **AsyncLongEvent**

@SCJAllowed(javax.safetycritical.annotate.Level.LEVEL_1)
public class AsyncLongEvent **extends** javax.realtime.AbstractAsyncEvent

In SCJ only aperiodic events are visible at the application level, and they are created only by the \scj infrastructure. Hence, constructors are hidden from public view. Handlers are attached to events by the infrastructure. Consequently the related methods are not visible. This class differs from AsyncEvent 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.LEVEL_1)
public void fire(**long** value)

fire this event, i.e., releases all the handlers that were added to this event.

value — is the data to be passed to the released event handler(s).

Memory behavior: Does not allocate memory. Does not allow this to escape local variables.

D.2.9 CLASS **AsyncLongEventHandler**

@SCJAllowed(javax.safetycritical.annotate.Level.LEVEL_0)
public class AsyncLongEventHandler

extends javax.realtime.AbstractAsyncEventHandler

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
public void handleAsyncEvent(**long** data)

This method must be overridden by the application to provide the handling code.

data — is the data that was passed when the associated event was fired.

D.2.10 CLASS **BoundAsyncEventHandler**

@SCJAllowed
public class BoundAsyncEventHandler

extends javax.realtime.AsyncEventHandler

The BoundAsyncEventHandler class is not directly available to the safety-critical Java application developers. Hence none of its methods or constructors are publicly available.

D.2.11 CLASS **BoundAsyncLongEventHandler**

@SCJAllowed
public class BoundAsyncLongEventHandler

extends javax.realtime.AsyncLongEventHandler

The BoundAsyncLongEventHandler class is not directly available to the safety-critical Java application developers. Hence none of its methods or constructors are publicly available. This class differs from BoundAsyncEventHandler in that when it is fired, a long integer is provided for use by the released event handler(s).

D.2.12 CLASS Clock

@SCJAllowed
public abstract class Clock **extends** java.lang.Object

A clock marks the passing of time. It has a concept of now that can be queried through Clock.getTime, and it can have events queued on it which will be fired 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 other clocks are not valid for any purpose that involves sleeping or waiting. They may, however, be used in the fire time and the period of OneShotTimer.

Constructors

@SCJAllowed @SCJRestricted(maySelfSuspend = false)
public Clock( )

Constructor for the abstract class.

Memory behavior: This constructor may allocate objects within the same MemoryArea that holds the implicit this argument.

Methods

@SCJAllowed @SCJRestricted(mayAllocate = false, maySelfSuspend = false)
protected abstract boolean drivesEvents( )

Returns true if and only if this Clock is able to trigger the execution of time-driven activities. Some user-defined clocks may be read-only, meaning the clock can be used to obtain timestamps, but the clock cannot be used to trigger the execution of events. If a clock that does not return drivesEvents equals true is used to configure a Timer or a sleep() request, an IllegalArgumentException will be thrown by the infrastructure. The default real-time clock does drive events.

returns true if the clock can drive events.

@SCJAllowed @SCJRestricted(maySelfSuspend = false)
public abstract javax.realtime.RelativeTime getEpochOffset( )

Returns the relative time of the offset of the epoch of this clock from the Epoch. For the real-time clock it will return a RelativeTime value equal to 0. An UnsupportedOperationException is thrown if the clock does not support the concept of date.

returns A newly allocated RelativeTime object in the current execution context with the offset past the Epoch for this clock. The returned object is associated with this clock.

Memory behavior: This constructor may allocate objects within the currently active MemoryArea.

@SCJAllowed @SCJRestricted(maySelfSuspend = false, mayAllocate = false)
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 @SCJRestricted(maySelfSuspend = false)
public abstract javax.realtime.RelativeTime getResolution( )

Gets the resolution of the clock, the nominal interval between ticks.

returns A newly allocated RelativeTime object in the current execution context representing the clock resolution. The returned object is associated with this clock.

@SCJAllowed @SCJRestricted(maySelfSuspend = false)
public abstract javax.realtime.RelativeTime getResolution(
  RelativeTime dest)

Gets the resolution of the clock, the nominal interval between ticks. The return value shall be associated with this clock. All relative time differences measured by this clock are approximately an integral multiple of the resolution.

dest — Return the relative time value in dest. If dest is null, a newly allocated RelativeTime object in the current execution context is returned. The returned object is associated with this clock.

returns dest is set to values representing the resolution of this. The returned object is associated with this clock.

@SCJAllowed @SCJRestricted(mayAllocate = false, maySelfSuspend = false)
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 an 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 which will be updated in place. The clock association of the dest parameter is ignored. 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 parameter, representing the current time, associated with this clock, or null if dest was null.

@SCJAllowed @SCJRestricted(maySelfSuspend = false)
public abstract 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.

Memory behavior: This constructor may allocate objects within the currently active MemoryArea.

@SCJAllowed(javax.safetycritical.annotate.Level.LEVEL_1)
@SCJRestricted(maySelfSuspend = false)
protected abstract void registerCallBack(AbsoluteTime time,
ClockCallBack clockEvent)

Code in the abstract base Clock class makes this call to the subclass. The method is expected to implement a mechanism that will invoke atTime in ClockCallBack at time time, and if this clock is subject to discontinuities, invoke ClockCallBack.discontinuity(javax.realtime.Clock, javax.realtime.RelativeTime) each time a clock discontinuity is detected. registerCallBack of this clock and invocations of atTime and resetTargetTime of clockEvent are protected by a clock specific lock.

time — The absolute time value on this clock at which ClockCallBack.atTime(Clock) should be invoked.

clockEvent — The object that should be notified at time. If clockEvent is null, unregister the current clock event.

@SCJAllowed(javax.safetycritical.annotate.Level.LEVEL_1)
@SCJRestricted(maySelfSuspend = false)
protected abstract boolean resetTargetTime(AbsoluteTime time)

Replace the target time being used by the ClockCallBack registered by registerCallBack(AbsoluteTime, ClockCallBack).

time — The new target time.

returns false if no ClockEvent is currently registered.

D.2.13 CLASS **DeregistrationException**

@SCJAllowed
public class DeregistrationException **extends** java.lang.RuntimeException

Kelvin added this on 2/8/11 because it is required by InterruptServiceRoutine. Did I get the right superclass?

Constructors

@SCJAllowed @SCJRestricted(maySelfSuspend = false)
public DeregistrationException( )

@SCJAllowed @SCJRestricted(maySelfSuspend = false)
public DeregistrationException(String description)

Kelvin wonders why this should be declared to allocate in immortal. This code was copied from MemoryScopeException. Why should that allocate in immortal?

Memory behavior: This constructor may allocate objects within the ImmortalMemory MemoryArea.

D.2.14 CLASS EventNotFoundException

@SCJAllowed(javax.safetycritical.annotate.Level.LEVEL_1)
public class EventNotFoundException **extends** java.lang.Exception

Useless description of EventNotFoundException.

Constructors

@SCJAllowed(javax.safetycritical.annotate.Level.LEVEL_1)
public EventNotFoundException( )

D.2.15 CLASS ExternalHappening

@SCJAllowed(javax.safetycritical.annotate.Level.LEVEL_1)
public class ExternalHappening
implements javax.realtime.Happening
extends java.lang.Object

A **javax.realtime.Happening** type for second level interrupt handling. This class provides a close parallel to the 1.0.2 Happening support. The primary differences are that

- this class and its subclasses are first-class entities in the RTSJ, not buried in the implementation and identified only by a String name;
- they include the Happening.trigger(int) method that allows a signal to be explicitly triggered by Java code, and at the implementation's option, a native code function that permits native application code to trigger the signal; and
- asynchronous events are attached to the signal instead of having happenings attached to asynchronous events.

A happenings may be assigned a unique name by the application, or the system will assign a name when they are not provided. The name space for system names is all strings beginning with javax.realtime.Happening.

Constructors

@SCJAllowed(javax.safetycritical.annotate.Level.LEVEL_1)
public ExternalHappening(String name)

Create an new EventHappening.

name — of the EventHappening

Methods

@SCJAllowed(javax.safetycritical.annotate.Level.LEVEL_1) @Override
public void attach(AbstractAsyncEvent event)
throws java.lang.IllegalArgumentException

@SCJAllowed(javax.safetycritical.annotate.Level.LEVEL_1)
@SCJRestricted(mayAllocate = false,
maySelfSuspend = false,
{javax.safetycritical.annotate.Phase.ALL})
public void deregister( )

Unregister this signal so that it cannot be triggered.

Throws DeregistrationException when this signal is not registered

@SCJAllowed(javax.safetycritical.annotate.Level.LEVEL_1) @Override
public boolean deregisterName( )

@SCJAllowed(javax.safetycritical.annotate.Level.LEVEL_1)
public static javax.realtime.ExternalHappening get(**int** id)

Get the external event corresponding to a given id.

id — of a registered signal

returns the signal corresponding to id.

@SCJAllowed(javax.safetycritical.annotate.Level.LEVEL_1)
@SCJRestricted(mayAllocate = false,
maySelfSuspend = false,
{javax.safetycritical.annotate.Phase.ALL})
public final int getId( )

Get the id of this signal.

returns the id of this signal.

```
@SCJAllowed(javax.safetycritical.annotate.Level.LEVEL_1)
public static int getId(String name)
```

Get the ID of a registered signal.

name — of the signal for which to search

returns the ID of the signal named by name

```
@SCJAllowed(javax.safetycritical.annotate.Level.LEVEL_1)
@SCJRestricted(mayAllocate = false,
   maySelfSuspend = false,
   {javax.safetycritical.annotate.Phase.ALL})
public java.lang.String getName( )
```

Get the name of this signal.

returns the name of this signal.

```
@SCJAllowed(javax.safetycritical.annotate.Level.LEVEL_1) @Override
public boolean isHappening(String name)
```

```
@SCJAllowed(javax.safetycritical.annotate.Level.LEVEL_1)
@SCJRestricted(mayAllocate = false,
   maySelfSuspend = false,
   {javax.safetycritical.annotate.Phase.ALL})
public boolean isRegistered( )
```

Indicates if the happening is registered or not.

returns true when the happening is registered

```
@SCJAllowed(javax.safetycritical.annotate.Level.LEVEL_1)
public javax.realtime.ExternalHappeningDispatcher register(
  ExternalHappeningDispatcher dispatcher)
  throws javax.realtime.RegistrationException
```

Register this signal with the given manager so that it can be triggered, thereby deregistering it from its previous dispatcher.

dispatcher — which will dispatch the trigger event.

returns the previous dispatcher

Throws RegistrationException when this signal is already registered

@SCJAllowed(javax.safetycritical.annotate.Level.LEVEL_1) @Override
public boolean registerName( )

@SCJAllowed(javax.safetycritical.annotate.Level.LEVEL_1)
public static void trigger(**int** id)

Release the manager for the signal identified by the given integer.

id — of a registered signal

@SCJAllowed(javax.safetycritical.annotate.Level.LEVEL_1) @Override
public void unbind(AbstractAsyncEvent event)
throws javax.realtime.EventNotFoundException

D.2.16 CLASS **ExternalHappeningDispatcher**

@SCJAllowed(javax.safetycritical.annotate.Level.LEVEL_1)
public abstract class ExternalHappeningDispatcher

extends javax.realtime.HappeningDispatcher

This class provides a means of dispatching a set of **javax.realtime.Happening** . The application must provide a RealtimeThread to perform this task. The application thread calls either the **takeControl()** or the **takeControlInterruptable()** methods. That method calls process() each time the signal is triggered and returns when the last **javax.realtime.ExternalHappening** is unregistered.

Constructors

@SCJAllowed(javax.safetycritical.annotate.Level.LEVEL_1)
public ExternalHappeningDispatcher(**int** size)

Create a new dispatcher.

size — gives the maximum number of outstanding trigger requests.

Methods

@SCJAllowed(javax.safetycritical.annotate.Level.LEVEL_1)
protected abstract void dispath(ExternalHappening happening)

Actually dispatch the **javax.realtime.ExternalHappening** . This can be overridden in a subclass to provide for more sophisticated dispatching.

happening — to dispatch

@SCJAllowed(javax.safetycritical.annotate.Level.LEVEL_1)
public static
javax.realtime.ExternalHappeningDispatcher getDefaultExternalHappeningDispatcher( )

This provides a means of obtaining the system provided event manager so that new events can be added to it.

returns the default event manager.

@SCJAllowed(javax.safetycritical.annotate.Level.LEVEL_1) @Override
public void takeControl( )

The application supplies a Schedulable object to the signal using this method. The happenings registered with this manager use the calling task to dispatch the happenings, e.g., by firing a happenings events. The method does not return until the last signal is deregistered.

TakeControl behaves effectively as if it were implemented as follows:

```
while (hasRegisteredHappening)
{
  waitForTrigger or for this signal to be unregistered
  if isRegistered
    process
  else
   return
}
```

A signal controlled by an application Scedulable can only be attached to AsyncEvents that are stored in the signal object. The method **takeControlInterruptable()** does not return until the signal is deregistered, or process() throws an exception.

Throws java.lang.IllegalStateException thrown when this signal is already controlled by a Schedulable, or the calling Schedulable's current memory area is not the memory area containing this signal, or no signal is not registered.

@SCJAllowed(javax.safetycritical.annotate.Level.LEVEL_1) @Override
public synchronized void takeControlInterruptable( )

The application supplies a Schedulable object to the signal using this method. The happenings registered with this manager use the calling task to dispatch the happenings, e.g., by firing a happenings events. The method does not return until the last signal is deregistered.

TakeControl behaves effectively as if it were implemented as follows:

```
while (hasRegisteredHappening)
{
  waitForTrigger or for this signal to be unregistered
  if isRegistered
    process
  else
   return
}
```

A signal controlled by an application Scedulable can only be attached to AsyncEvents that are stored in the signal object. The method **javax.realtime.External-HappeningDispatchertakeControlInterruptable** does not return until the signal is deregistered, or process() throws an exception.

Throws java.lang.IllegalStateException thrown when this signal is already controlled by a Schedulable, or the calling Schedulable's current memory area is not the memory area containing this signal, or no signal is not registered.

D.2.17 CLASS HappeningDispatcher

@SCJAllowed(javax.safetycritical.annotate.Level.LEVEL_1)
public abstract class HappeningDispatcher **extends** java.lang.Object

HappeningDispatcher provides a means of dispatching a set of EventHappenings. The application must provide a RealtimeThread to perform this task. The application thread calls either the **takeControl()** or the **takeControlInterruptable()** methods. That method calls process() each time the signal is triggered and returns when the last **javax.realtime.Happening** is unregistered.

Constructors

@SCJAllowed(javax.safetycritical.annotate.Level.LEVEL_1)
public HappeningDispatcher( )

Methods

@SCJAllowed(javax.safetycritical.annotate.Level.LEVEL_1)
public abstract void takeControl( )

The application supplies a Schedulable object to the signal using this method. The happenings registered with this manager use the calling task to dispatch the happenings, e.g., by firing a happenings events. The method does not return until the last signal is deregistered.

TakeControl behaves effectively as if it were implemented as follows:

```
while (hasRegisteredHappening)
{
  waitForTrigger or for this signal to be unregistered
  if isRegistered
    process
  else
   return
}
```

A signal controlled by an application Scedulable can only be attached to AsyncEvents that are stored in the signal object. The method **takeControlInterruptable()** does not return until the signal is deregistered, or process() throws an exception.

Throws java.lang.IllegalStateException thrown when this signal is already controlled by a Schedulable, or the calling Schedulable's current memory area is not the memory area containing this signal, or no signal is not registered.

@SCJAllowed(javax.safetycritical.annotate.Level.LEVEL_1)
public abstract void takeControlInterruptable( )

The application supplies a Schedulable object to the signal using this method. The happenings registered with this manager use the calling task to dispatch the happenings, e.g., by firing a happenings events. The method does not return until the last signal is deregistered.

TakeControl behaves effectively as if it were implemented as follows:

```
while (hasRegisteredHappening)
{
  waitForTrigger or for this signal to be unregistered
  if isRegistered
    process
  else
   return
}
```

A signal controlled by an application Scedulable can only be attached to AsyncEvents that are stored in the signal object. The method **javax.realtime.Happening-DispatchertakeControlInterruptable** does not return until the signal is deregistered, or process() throws an exception.

Throws java.lang.IllegalStateException thrown when this signal is already controlled by a Schedulable, or the calling Schedulable's current memory area is not the memory area containing this signal, or no signal is not registered.

D.2.18 CLASS HighResolutionTime

@SCJAllowed
public abstract class HighResolutionTime
  implements java.lang.Comparable
  extends java.lang.Object

Class HighResolutionTime is the base class for AbsoluteTime, and RelativeTime. Time can be expressed with nanosecond accuracy. This class is never used directly: it is abstract and has no public constructor. Instead, one of its subclasses AbsoluteTime or RelativeTime should be used.

Methods

@SCJAllowed @SCJRestricted(maySelfSuspend = false)
public int compareTo(HighResolutionTime time)

Compares this HighResolutionTime with the specified HighResolutionTimetime.

time — compares with the time of this.

@SCJAllowed @SCJRestricted(maySelfSuspend = false)
public int compareTo(Object object)

Compares this HighResolutionTime with the specified object.

object — compares with the time of this.

@SCJAllowed @SCJRestricted(maySelfSuspend = false, mayAllocate = false)
public boolean equals(HighResolutionTime time)

Returns true if the argument object has the same type and values as this.

time — Value compared to this.

returns true if the parameter object is of the same type and has the same values as this.

@SCJAllowed @SCJRestricted(maySelfSuspend = false, mayAllocate = false)
public boolean equals(Object object)

Returns true if the argument object has the same type and values as this.

object — Value compared to this.

returns true if the parameter object is of the same type and has the same values as this.

@SCJAllowed @SCJRestricted(maySelfSuspend = false)
public javax.realtime.Clock getClock( )

returns A reference to the clock associated with this.

@SCJAllowed @SCJRestricted(maySelfSuspend = false)
public final long getMilliseconds( )

returns The milliseconds component of the time represented by this.

@SCJAllowed @SCJRestricted(maySelfSuspend = false)
public final int getNanoseconds( )

returns The nanoseconds component of the time represented by this.

@SCJAllowed @SCJRestricted(maySelfSuspend = false, mayAllocate = false)
public int hashCode( )

Returns a hash code for this object in accordance with the general contract of Object.hashCode.

returns The hashcode value for this instance.

@SCJAllowed @SCJRestricted(maySelfSuspend = false)
public void set(**long** millis)

Sets the millisecond component of this to the given argument, and the nanosecond component of this to 0.

millis — This value shall be the value of the millisecond component of this at the completion of the call.

@SCJAllowed @SCJRestricted(maySelfSuspend = false, mayAllocate = false)
public void set(**long** millis, **int** nanos)

Sets the millisecond and nanosecond components of this.

millis — The desired value for the millisecond component of this at the completion of the call. The actual value is the result of parameter normalization.

nanos — The desired value for the nanosecond component of this at the completion of the call. The actual value is the result of parameter normalization.

@SCJAllowed @SCJRestricted(maySelfSuspend = false)
public void set(HighResolutionTime time)

Change the value represented by this to that of the given time.

time — The new value for this.

@SCJAllowed(javax.safetycritical.annotate.Level.LEVEL_2)
public static void 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.

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.

D.2.19 CLASS **IllegalAssignmentError**

@SCJAllowed
public class IllegalAssignmentError
implements java.io.Serializable
extends java.lang.Error

Constructors

@SCJAllowed @SCJRestricted(maySelfSuspend = false)
public IllegalAssignmentError( )

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).

Memory behavior: This constructor may allocate objects within the currently active MemoryArea.

@SCJAllowed @SCJRestricted(maySelfSuspend = false)
public IllegalAssignmentError(String description)

Shall not copy "this" to any instance or static field. The scope containing the msg argument must enclose the scope containing "this". Otherwise, an Illegal-AssignmentError will be thrown.

Allocates an application- and implementation-dependent amount of memory in the current scope (to represent stack backtrace).

Memory behavior: This constructor may allocate objects within the currently active MemoryArea.

This constructor requires that the "description" argument reside in a scope that encloses the scope of the "this" argument.

D.2.20 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.

D.2.21 CLASS **InaccessibleAreaException**

@SCJAllowed
public class InaccessibleAreaException
  implements java.io.Serializable
  extends java.lang.RuntimeException

Useless description of InacessibleAreaException.

Constructors

@SCJAllowed @SCJRestricted(maySelfSuspend = false)
public InaccessibleAreaException( )

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).

Memory behavior: This constructor may allocate objects within the currently active MemoryArea.

@SCJAllowed @SCJRestricted(maySelfSuspend = false)
public InaccessibleAreaException(String description)

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).

Memory behavior: This constructor may allocate objects within the currently active MemoryArea.

This constructor requires that the "msg" argument reside in a scope that encloses the scope of the "this" argument.

D.2.22 CLASS **InterruptServiceRoutine**

@SCJAllowed(javax.safetycritical.annotate.Level.LEVEL_1)
public abstract class InterruptServiceRoutine **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.

Methods

@SCJAllowed(javax.safetycritical.annotate.Level.LEVEL_1)
public static javax.realtime.InterruptServiceRoutine getISR(
  int interrupt)

returns the ISR registered with the given interrupt. Null is returned if nothing is registered.

@SCJAllowed(javax.safetycritical.annotate.Level.LEVEL_1)
public static javax.realtime.AffinitySet getInterruptAffinity(
  int InterruptId)

Every interrupt has an affinity that indicates which processors might service a hardware interrupt request. The returned set is preallocated and resides in immortal memory.

returns The affinity set of the processors.

Throws IllegalArgument if unsupported InterruptId

@SCJAllowed(javax.safetycritical.annotate.Level.LEVEL_1)
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 IllegalArgument if unsupported InterruptId

@SCJAllowed(javax.safetycritical.annotate.Level.LEVEL_1)
public final java.lang.String getName( )

Get the name of this interrupt service routine.

returns the name of this interrupt service routine.

@SCJAllowed(javax.safetycritical.annotate.Level.LEVEL_1)
@SCJRestricted({javax.safetycritical.annotate.Phase.
   INTERRUPT_SERVICE_ROUTINE}, maySelfSuspend = false)
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. Unless the overridden method is synchronized, the infrastructure shall provide no synchronization for the execution of this method.

D.2.23 CLASS **LTMemory**

@SCJAllowed
public class LTMemory **extends** javax.realtime.ScopedMemory

This class can not be instantiated in SCJ. It is subclassed by MissionMemory and PrivateMemory. It has no visible methods.

D.2.24 CLASS **MemoryAccessError**

@SCJAllowed
public class MemoryAccessError
  implements java.io.Serializable
  extends java.lang.RuntimeException

Constructors

@SCJAllowed @SCJRestricted(maySelfSuspend = false)
public MemoryAccessError( )

Memory behavior: This constructor may allocate objects within the currently active MemoryArea.

@SCJAllowed @SCJRestricted(maySelfSuspend = false)
public MemoryAccessError(String description)

Memory behavior: This constructor may allocate objects within the currently active MemoryArea.

This constructor requires that the "msg" argument reside in a scope that encloses the scope of the "this" argument.

D.2.25 CLASS **MemoryArea**

@SCJAllowed
public abstract class MemoryArea
  implements javax.realtime.AllocationContext
  extends java.lang.Object

All allocation contexts are implemented by memory areas. This is the base-level class for all memory areas.

Methods

@SCJAllowed @SCJRestricted(maySelfSuspend = false)
public static javax.realtime.MemoryArea getMemoryArea(Object object)

returns the memory area in which object is allocated,

@SCJAllowed @SCJRestricted(maySelfSuspend = false)
public abstract long memoryConsumed( )

returns the memory consumed in this memory area.

@SCJAllowed @SCJRestricted(maySelfSuspend = false)
public abstract long memoryRemaining( )

returns the memory remaining in this memory area.

@SCJAllowed @SCJRestricted(maySelfSuspend = false)
public java.lang.Object newArray(**Class**<> type, **int** size)
throws javax.realtime.InaccessibleAreaException

This method creates an object of type type in the same memory area that contains this.

type — is the type of the array element for the returned array.

returns a new array of type type with size n objects allocated in this memory area.

@SCJAllowed @SCJRestricted(maySelfSuspend = false)
public java.lang.Object newInstance(**Class**<> type)
throws java.lang.IllegalArgumentException,
java.lang.InstantiationException,
java.lang.OutOfMemoryError,
javax.realtime.InaccessibleAreaException

This method creates an object of type type in this memory area.

type — is the type of the object returned.

returns a new object of type type

Memory behavior: This constructor may allocate objects within the same MemoryArea that holds the argument named "this.area".

@SCJAllowed @SCJRestricted(maySelfSuspend = false)
public abstract long size( )

The size of a memory area is always equal to the {memoryConsumed() + memoryRemaining()}.

returns the total size of this memory area.

D.2.26 CLASS **MemoryInUseException**

@SCJAllowed
public class MemoryInUseException **extends** java.lang.RuntimeException

Constructors

@SCJAllowed @SCJRestricted(maySelfSuspend = false)
public MemoryInUseException( )

@SCJAllowed @SCJRestricted(maySelfSuspend = false)
public MemoryInUseException(String description)

Memory behavior: This constructor may allocate objects within the ImmortalMemory MemoryArea.

D.2.27 CLASS **MemoryParameters**

@SCJAllowed
public class MemoryParameters
  implements java.lang.Cloneable
  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 NO_MAX

Constructors

@SCJAllowed
public MemoryParameters(**long** maxMemoryArea, **long** maxImmortal)

Create a MemoryParameters object with the given maximum values.

maxMemoryArea — 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 other than positive. zero, or NO_MAX is passed as the value of maxMemoryArea or maxImmortal.

D.2.28 CLASS **MemoryScopeException**

@SCJAllowed
public class MemoryScopeException **extends** java.lang.RuntimeException

Constructors

@SCJAllowed @SCJRestricted(maySelfSuspend = false)
public MemoryScopeException( )

@SCJAllowed @SCJRestricted(maySelfSuspend = false)
public MemoryScopeException(String description)

Memory behavior: This constructor may allocate objects within the ImmortalMemory MemoryArea.

D.2.29 CLASS **NoHeapRealtimeThread**

@SCJAllowed(javax.safetycritical.annotate.Level.LEVEL_2)
public class NoHeapRealtimeThread **extends** javax.realtime.RealtimeThread

NoHeapRealtimeThreads} cannot be directly created by the SCJ application. However, they are needed by the infrastructure to support ManagedThreads at Level 2.

Constructors

@SCJAllowed(javax.safetycritical.annotate.Level.LEVEL_2)
public NoHeapRealtimeThread(SchedulingParameters schedule,
  MemoryArea area)

TBD: do we use this constructor, which expects a MemoryArea argument?

@SCJAllowed(javax.safetycritical.annotate.Level.LEVEL_2)
public NoHeapRealtimeThread(SchedulingParameters schedule,
  ReleaseParameters release)

TBD: do we use this constructor, which expects a ReleaseParameters argument?

Methods

@SCJAllowed(javax.safetycritical.annotate.Level.LEVEL_2)
@SCJRestricted(mayAllocate = false, maySelfSuspend = false)
public void start( )

Creation of thread may block, but start shall not block.

D.2.30 CLASS **POSIXRealtimeSignal**

@SCJAllowed(javax.safetycritical.annotate.Level.LEVEL_1)
public class POSIXRealtimeSignal
  implements javax.realtime.Happening
  extends javax.realtime.AsyncLongEvent

Useless description of POSIXRealtimeSignal.

Methods

@SCJAllowed(javax.safetycritical.annotate.Level.LEVEL_1) @Override
public void attach(AbstractAsyncEvent event)
  throws java.lang.IllegalArgumentException

@SCJAllowed(javax.safetycritical.annotate.Level.LEVEL_1)
@SCJRestricted(mayAllocate = false,
   maySelfSuspend = false,
   {javax.safetycritical.annotate.Phase.ALL})
public void deregister( )

Unregister this signal so that it cannot be triggered.

Throws DeregistrationException when this signal is not registered

@SCJAllowed(javax.safetycritical.annotate.Level.LEVEL_1) @Override
public boolean deregisterName( )

@SCJAllowed(javax.safetycritical.annotate.Level.LEVEL_1)
public static javax.realtime.POSIXRealtimeSignal get(**int** id)

Get the realtime signal corresponding to a given id.

id — of a registered signal

returns the signal corresponding to id.

@SCJAllowed(javax.safetycritical.annotate.Level.LEVEL_1)
@SCJRestricted(mayAllocate = false,
  maySelfSuspend = false,
  {javax.safetycritical.annotate.Phase.ALL})
public int getId( )

@SCJAllowed(javax.safetycritical.annotate.Level.LEVEL_1)
@SCJRestricted(mayAllocate = false,
  maySelfSuspend = false,
  {javax.safetycritical.annotate.Phase.ALL})
public java.lang.String getName( )

Get the name of this signal.

returns the name of this signal.

@SCJAllowed(javax.safetycritical.annotate.Level.LEVEL_1)
public long getNextValue( )

@SCJAllowed(javax.safetycritical.annotate.Level.LEVEL_1) @Override
public boolean isHappening(String name)

@SCJAllowed(javax.safetycritical.annotate.Level.LEVEL_1)
@SCJRestricted(mayAllocate = false,
  maySelfSuspend = false,
  {javax.safetycritical.annotate.Phase.ALL})
public boolean isRegistered( )

@SCJAllowed(javax.safetycritical.annotate.Level.LEVEL_1)
public javax.realtime.POSIXRealtimeSignalDispatcher register(
 POSIXRealtimeSignalDispatcher dispatcher)
 throws javax.realtime.RegistrationException

Register this signal with the given manager so that it can be triggered, thereby deregistering it from its previous dispatcher.

dispatcher — which will dispatch the trigger event.

returns the previous dispatcher

Throws RegistrationException when this signal is already registered

@SCJAllowed(javax.safetycritical.annotate.Level.LEVEL_1) @Override
public boolean registerName( )

@SCJAllowed(javax.safetycritical.annotate.Level.LEVEL_1)
public static void trigger(**int** id, **long** value)

Release the manager for the Signal identified by the given integer. The id range for Signals is distinct from that of ExternalEvents. This method is provided for simulating external events.

id — of a registered signal

@SCJAllowed(javax.safetycritical.annotate.Level.LEVEL_1) @Override
public void unbind(AbstractAsyncEvent event)
throws javax.realtime.EventNotFoundException

D.2.31 CLASS **POSIXRealtimeSignalDispatcher**

@SCJAllowed(javax.safetycritical.annotate.Level.LEVEL_1)
public abstract class POSIXRealtimeSignalDispatcher

extends javax.realtime.HappeningDispatcher

This class provides a means of dispatching a set of Happenings. The application must provide a RealtimeThread to perform this task. The application thread calls either the **takeControl()** or the **takeControlInterruptable()** methods. That method calls process() each time the signal is triggered and returns when the last ExternalHappening is unregistered.

Constructors

@SCJAllowed(javax.safetycritical.annotate.Level.LEVEL_1)
public POSIXRealtimeSignalDispatcher(**int** size)

Create a new dispatcher.

size — gives the maximum number of outstanding trigger requests.

Methods

@SCJAllowed(javax.safetycritical.annotate.Level.LEVEL_1)
protected abstract void dispatch(POSIXRealtimeSignal signal)

Actually dispatch the **javax.realtime.POSIXRealtimeSignal** . This can be overridden in a subclass to provide for more sophisticated dispatching.

signal — to dispatch

@SCJAllowed(javax.safetycritical.annotate.Level.LEVEL_1)
public static
javax.realtime.POSIXRealtimeSignalDispatcher getDefaultPOSIXRealtimeSignalDispatcher( )

This provides a means of obtaining the system provided event manager so that new events can be added to it.

returns the default event manager.

@SCJAllowed(javax.safetycritical.annotate.Level.LEVEL_1) @Override
public void takeControl( )

The application supplies a Schedulable object to the signal using this method. The happenings registered with this manager use the calling task to dispatch the happenings, e.g., by firing a happenings events. The method does not return until the last signal is deregistered.

TakeControl behaves effectively as if it were implemented as follows:

```
while (hasRegisteredHappening)
{
  waitForTrigger or for this signal to be unregistered
  if isRegistered
    process
  else
   return
}
```

A signal controlled by an application Scedulable can only be attached to AsyncEvents that are stored in the signal object. The method **takeControlInterruptable()** does not return until the signal is deregistered, or process() throws an exception.

Throws java.lang.IllegalStateException thrown when this signal is already controlled by a Schedulable, or the calling Schedulable's current memory area is not the memory area containing this signal, or no signal is not registered.

@SCJAllowed(javax.safetycritical.annotate.Level.LEVEL_1) @Override
public void takeControlInterruptable( )

The application supplies a Schedulable object to the signal using this method. The happenings registered with this manager use the calling task to dispatch the happenings, e.g., by firing a happenings events. The method does not return until the last signal is deregistered.

TakeControl behaves effectively as if it were implemented as follows:

```
while (hasRegisteredHappening)
{
  waitForTrigger or for this signal to be unregistered
  if isRegistered
    process
  else
   return
}
```

A signal controlled by an application Scedulable can only be attached to AsyncEvents that are stored in the signal object. The method **javax.realtime.POSIXRealtime-SignalDispatchertakeControlInterruptable** does not return until the signal is deregistered, or process() throws an exception.

Throws java.lang.IllegalStateException thrown when this signal is already controlled by a Schedulable, or the calling Schedulable's current memory area is not the memory area containing this signal, or no signal is not registered.

@SCJAllowed(javax.safetycritical.annotate.Level.LEVEL_1)
public void trigger(POSIXRealtimeSignal signal, **long** value)

Queue the event for dispatching by this manager. This should only be called from @{link EventHappening#trigger()}.

signal — the event that needs to be dispatched

value — to be provided to the event handler for this real-time signal

D.2.32 CLASS **POSIXSignal**

@SCJAllowed(javax.safetycritical.annotate.Level.LEVEL_1)
public class POSIXSignal
  implements javax.realtime.Happening
  extends javax.realtime.AsyncEvent

Useless description of POSIXSignal.

Methods

@SCJAllowed(javax.safetycritical.annotate.Level.LEVEL_1) @Override
public void attach(AbstractAsyncEvent event)
  throws java.lang.IllegalArgumentException

@SCJAllowed(javax.safetycritical.annotate.Level.LEVEL_1)
@SCJRestricted(mayAllocate = false,
  maySelfSuspend = false,
  {javax.safetycritical.annotate.Phase.ALL})
public void deregister( )

Unregister this signal so that it cannot be triggered.

Throws DeregistrationException when this signal is not registered

@SCJAllowed(javax.safetycritical.annotate.Level.LEVEL_1) @Override
public boolean deregisterName( )

@SCJAllowed(javax.safetycritical.annotate.Level.LEVEL_1)
public static javax.realtime.POSIXSignal get(**int** id)

Get the signal corresponding to a given id.

id — of a registered signal

returns the signal corresponding to id or null.

@SCJAllowed(javax.safetycritical.annotate.Level.LEVEL_1)
@SCJRestricted(mayAllocate = false,
  maySelfSuspend = false,
  {javax.safetycritical.annotate.Phase.ALL})
public int getId( )

@SCJAllowed(javax.safetycritical.annotate.Level.LEVEL_1)
@SCJRestricted(mayAllocate = false,
  maySelfSuspend = false,
  {javax.safetycritical.annotate.Phase.ALL})
public java.lang.String getName( )

Get the name of this signal.

returns the name of this signal.

@SCJAllowed(javax.safetycritical.annotate.Level.LEVEL_1) @Override
public boolean isHappening(String name)

```
@SCJAllowed(javax.safetycritical.annotate.Level.LEVEL_1)
@SCJRestricted(mayAllocate = false,
   maySelfSuspend = false,
   {javax.safetycritical.annotate.Phase.ALL})
public boolean isRegistered( )
```

Indicates if the happening is registered or not.

returns true when the happening is registered

```
@SCJAllowed(javax.safetycritical.annotate.Level.LEVEL_1)
public javax.realtime.POSIXSignalDispatcher register(
  POSIXSignalDispatcher dispatcher)
  throws javax.realtime.RegistrationException
```

Register this signal with the given manager so that it can be triggered, thereby deregistering it from its previous dispatcher.

dispatcher — which will dispatch the trigger event.

returns the previous dispatcher

Throws RegistrationException when this signal is already registered

```
@SCJAllowed(javax.safetycritical.annotate.Level.LEVEL_1) @Override
public boolean registerName( )
```

```
@SCJAllowed(javax.safetycritical.annotate.Level.LEVEL_1)
public static void trigger(int id)
```

Release the manager for the Signal identified by the given integer. The id range for Signals is distinct from that of ExternalEvents. This method is provided for simulating external events.

id — of a registered signal

```
@SCJAllowed(javax.safetycritical.annotate.Level.LEVEL_1) @Override
public void unbind(AbstractAsyncEvent event)
  throws javax.realtime.EventNotFoundException
```

D.2.33 CLASS **POSIXSignalDispatcher**

```
@SCJAllowed(javax.safetycritical.annotate.Level.LEVEL_1)
public abstract class POSIXSignalDispatcher
```

extends javax.realtime.HappeningDispatcher

This class provides a means of dispatching a set of Happening. The application must provide a RealtimeThread to perform this task. The application thread calls either the **takeControl()** or the **takeControlInterruptable()** methods. That method calls process() each time the signal is triggered and returns when the last {ExternalHappening is unregistered.

Constructors

@SCJAllowed(javax.safetycritical.annotate.Level.LEVEL_1)
public POSIXSignalDispatcher(**int** size)

Create a new dispatcher.

size — gives the maximum number of outstanding trigger requests.

Methods

@SCJAllowed(javax.safetycritical.annotate.Level.LEVEL_1)
protected abstract void dispath(POSIXSignal signal)

Actually dispatch the **javax.realtime.POSIXSignal** . This can be overridden in a subclass to provide for more sophisticated dispatching.

signal — to dispatch

@SCJAllowed(javax.safetycritical.annotate.Level.LEVEL_1)
public static
javax.realtime.POSIXSignalDispatcher getDefaultPOSIXSignalDispatcher( )

This provides a means of obtaining the system provided event manager so that new events can be added to it.

returns the default event manager.

@SCJAllowed(javax.safetycritical.annotate.Level.LEVEL_1) @Override
public void takeControl( )

The application supplies a Schedulable object to the signal using this method. The happenings registered with this manager use the calling task to dispatch the happenings, e.g., by firing a happenings events. The method does not return until the last signal is deregistered.

TakeControl behaves effectively as if it were implemented as follows:

```
while (hasRegisteredHappening)
{
  waitForTrigger or for this signal to be unregistered
  if isRegistered
    process
  else
   return
}
```

A signal controlled by an application Scedulable can only be attached to AsyncEvents that are stored in the signal object. The method **takeControlInterruptable()** does not return until the signal is deregistered, or process() throws an exception.

Throws java.lang.IllegalStateException thrown when this signal is already controlled by a Schedulable, or the calling Schedulable's current memory area is not the memory area containing this signal, or no signal is not registered.

@SCJAllowed(javax.safetycritical.annotate.Level.LEVEL_1) @Override
public void takeControlInterruptable( )

The application supplies a Schedulable object to the signal using this method. The happenings registered with this manager use the calling task to dispatch the happenings, e.g., by firing a happenings events. The method does not return until the last signal is deregistered.

TakeControl behaves effectively as if it were implemented as follows:

```
while (hasRegisteredHappening)
{
  waitForTrigger or for this signal to be unregistered
  if isRegistered
    process
  else
   return
}
```

A signal controlled by an application Scedulable can only be attached to AsyncEvents that are stored in the signal object. The method **javax.realtime.POSIXSignal-DispatchertakeControlInterruptable** does not return until the signal is deregistered, or process() throws an exception.

Throws java.lang.IllegalStateException thrown when this signal is already controlled by a Schedulable, or the calling Schedulable's current memory area is not the memory area containing this signal, or no signal is not registered.

@SCJAllowed(javax.safetycritical.annotate.Level.LEVEL_1)
public void trigger(POSIXSignal signal)

Queue the event for dispatching by this manager. This should only be called from @{link EventHappening#trigger()}.

signal — the event that needs to be dispatched

D.2.34 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 @SCJRestricted(maySelfSuspend = false)
public PeriodicParameters(HighResolutionTime start, RelativeTime period)

Constructs a new PeriodicParameters object within the current memory area.

start — is the time of the first release of the associated schedulable object relative to the start of the mission. If the start time is in the past, the first release shall occur immediately. A null value defaults to an offset of zero milliseconds. An absolute start time is also measured relative to the start of the mission.

period — is the time between each release of the associated schedulable object. The default deadline is the same value as the period. The default handler is null.

Throws IllegalArgumentException if period is null.

@SCJAllowed @SCJRestricted(maySelfSuspend = false)
public PeriodicParameters(HighResolutionTime start,
RelativeTime period,
RelativeTime deadline,
AsyncEventHandler handler)

Construct a new PeriodicParameters object within the current memory area.

start — is time of the first release of the associated schedulable 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.

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

Throws IllegalArgumentException if period is null.

D.2.35 CLASS **PhysicalMemoryManager**

@SCJAllowed(javax.safetycritical.annotate.Level.LEVEL_1)
public final class PhysicalMemoryManager **extends** java.lang.Object

Fields

@SCJAllowed(javax.safetycritical.annotate.Level.LEVEL_1)
public static final javax.realtime.PhysicalMemoryName DEVICE

@SCJAllowed(javax.safetycritical.annotate.Level.LEVEL_1)
public static final javax.realtime.PhysicalMemoryName DMA

@SCJAllowed(javax.safetycritical.annotate.Level.LEVEL_1)
public static final javax.realtime.PhysicalMemoryName IO_PAGE

@SCJAllowed(javax.safetycritical.annotate.Level.LEVEL_1)
public static final javax.realtime.PhysicalMemoryName SHARED

D.2.36 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 4.6.5). 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 @SCJRestricted(maySelfSuspend = false)
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 @SCJRestricted(maySelfSuspend = false)
public int getPriority( )

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

D.2.37 CLASS **PriorityScheduler**

@SCJAllowed(javax.safetycritical.annotate.Level.LEVEL_1)
public 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(javax.safetycritical.annotate.Level.LEVEL_1)
@SCJRestricted(maySelfSuspend = false)
public int getMaxPriority( )

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

@SCJAllowed(javax.safetycritical.annotate.Level.LEVEL_1)
@SCJRestricted(maySelfSuspend = false)
public int getMinPriority( )

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

D.2.38 CLASS **ProcessorAffinityException**

@SCJAllowed(javax.safetycritical.annotate.Level.LEVEL_1)
public class ProcessorAffinityException **extends** java.lang.Exception

Constructors

@SCJAllowed(javax.safetycritical.annotate.Level.LEVEL_1)
public ProcessorAffinityException( )

@SCJAllowed(javax.safetycritical.annotate.Level.LEVEL_1)
public ProcessorAffinityException(String msg)

D.2.39 CLASS **RawMemory**

@SCJAllowed(javax.safetycritical.annotate.Level.LEVEL_0)
public final class RawMemory **extends** java.lang.Object

This class is the hub of a system that constructs special-purpose objects that access particular types and ranges of raw memory. This facility is supported by the registerRawIntegralAccessFactory and the create methods. Four raw-integral-access factories are supported: two for accessing memory (called IO_PORT_MAPPED and IO_MEMORY_MAPPED), one for accessing memory that can be used for DMA (called DMA_ACCESS) and the other for accesses to the memory (called MEM_ACCESS).

Fields

@SCJAllowed(javax.safetycritical.annotate.Level.LEVEL_0)
public static final javax.realtime.RawMemoryName DMA_ACCESS

The name indicating an area of raw memory which is accessable for DMA transfer.

@SCJAllowed(javax.safetycritical.annotate.Level.LEVEL_0)
public static final javax.realtime.RawMemoryName IO_MEM_MAPPED

The name indicating an area of raw memory which is used to access memory mapped IO registers.

@SCJAllowed(javax.safetycritical.annotate.Level.LEVEL_0)
public static final javax.realtime.RawMemoryName IO_PORT_MAPPED

The name indicating an area of raw memory which is used as port-mapped IO registers.

@SCJAllowed(javax.safetycritical.annotate.Level.LEVEL_0)
public static final javax.realtime.RawMemoryName MEM_ACCESS

The name indicating an area of raw memory.

Constructors

@SCJAllowed(javax.safetycritical.annotate.Level.LEVEL_0)
public RawMemory( )

Methods

@SCJAllowed(javax.safetycritical.annotate.Level.LEVEL_0)
@SCJRestricted({javax.safetycritical.annotate.Phase.
  INTERRUPT_SERVICE_ROUTINE}, mayAllocate = false, maySelfSuspend = false)
public static javax.realtime.RawByteArray createRawByteArrayInstance(
 RawMemoryName type,
 long base,
 long size)

Creates or finds an accessor object for accessing a byte array in raw memory.

type — is the required type of memory.

base — is the offset of the required array.

size — is the length of the array.

returns an accessor object from the raw memory access.

Throws AlignmentError if the base is not on the appropriate boundary.

Throws SizeOutOfBoundsException if the byte array falls in an invalid address range.

Throws MemoryTypeConflictException if base does not point to memory that matches the type served by this factory.

Throws OffsetOutOfBoundsException if the base is negative or greater than the size of the raw memory area.

@SCJAllowed(javax.safetycritical.annotate.Level.LEVEL_0)
@SCJRestricted({javax.safetycritical.annotate.Phase.
  INTERRUPT_SERVICE_ROUTINE}, mayAllocate = false, maySelfSuspend = false)
public static
javax.realtime.RawByteArrayRead createRawByteArrayReadInstance(
 RawMemoryName type,
 long base,
 long size)

Creates or finds an accessor object for accessing a byte array in raw memory.

type — is the required type of memory.

base — is the offset of the required array.

size — is the length of the array.

returns an accessor object from the raw memory access.

Throws AlignmentError if the base is not on the appropriate boundary.

Throws SizeOutOfBoundsException if the byte array falls in an invalid address range.

Throws MemoryTypeConflictException if base does not point to memory that matches the type served by this factory.

Throws OffsetOutOfBoundsException if the base is negative or greater than the size of the raw memory area.

```
@SCJAllowed(javax.safetycritical.annotate.Level.LEVEL_0)
@SCJRestricted({javax.safetycritical.annotate.Phase.
    INTERRUPT_SERVICE_ROUTINE}, mayAllocate = false, maySelfSuspend = false)
public static
javax.realtime.RawByteArrayWrite createRawByteArrayWriteInstance(
  RawMemoryName type,
  long base,
  long size)
```

Creates or finds an accessor object for accessing a byte array in raw memory.

type — is the required type of memory.

base — is the offset of the required array.

size — is the length of the array.

returns an accessor object from the raw memory access.

Throws AlignmentError if the base is not on the appropriate boundary.

Throws SizeOutOfBoundsException if the byte array falls in an invalid address range.

Throws MemoryTypeConflictException if base does not point to memory that matches the type served by this factory.

Throws OffsetOutOfBoundsException if the base is negative or greater than the size of the raw memory area.

```
@SCJAllowed(javax.safetycritical.annotate.Level.LEVEL_0)
@SCJRestricted({javax.safetycritical.annotate.Phase.
   INTERRUPT_SERVICE_ROUTINE}, mayAllocate = false, maySelfSuspend = false)
public static javax.realtime.RawByte createRawByteInstance(
  RawMemoryName type,
  long base)
```

Creates or finds an accessor object for accessing a byte of raw memory.

type — is the required type of memory.

base — is the offset of the required byte.

returns an accessor object from the raw memory access.

Throws AlignmentError if the base is not on the appropriate boundary.

Throws SizeOutOfBoundsException if the long array falls in an invalid address range.

Throws MemoryTypeConflictException if base does not point to memory that matches the type served by this factory.

Throws OffsetOutOfBoundsException if the base is negative or greater than the size of the raw memory area.

```
@SCJAllowed(javax.safetycritical.annotate.Level.LEVEL_0)
@SCJRestricted({javax.safetycritical.annotate.Phase.
   INTERRUPT_SERVICE_ROUTINE}, mayAllocate = false, maySelfSuspend = false)
public static javax.realtime.RawByteRead createRawByteReadInstance(
  RawMemoryName type,
  long base)
```

Creates or finds an accessor object for accessing a byte of raw memory.

type — is the required type of memory.

base — is the offset of the required byte.

returns an accessor object from the raw memory access.

Throws AlignmentError if the base is not on the appropriate boundary.

Throws SizeOutOfBoundsException if the long array falls in an invalid address range.

Throws MemoryTypeConflictException if base does not point to memory that matches the type served by this factory.

Throws OffsetOutOfBoundsException if the base is negative or greater than the size of the raw memory area.

```
@SCJAllowed(javax.safetycritical.annotate.Level.LEVEL_0)
@SCJRestricted({javax.safetycritical.annotate.Phase.
    INTERRUPT_SERVICE_ROUTINE}, mayAllocate = false, maySelfSuspend = false)
public static javax.realtime.RawByteWrite createRawByteWriteInstance(
  RawMemoryName type,
  long base)
```

Creates or finds an accessor object for accessing a byte of raw memory.

type — is the required type of memory.

base — is the offset of the required byte.

returns an accessor object from the raw memory access.

Throws AlignmentError if the base is not on the appropriate boundary.

Throws SizeOutOfBoundsException if the long array falls in an invalid address range.

Throws MemoryTypeConflictException if base does not point to memory that matches the type served by this factory.

Throws OffsetOutOfBoundsException if the base is negative or greater than the size of the raw memory area.

```
@SCJAllowed(javax.safetycritical.annotate.Level.LEVEL_0)
@SCJRestricted({javax.safetycritical.annotate.Phase.
    INTERRUPT_SERVICE_ROUTINE}, mayAllocate = false, maySelfSuspend = false)
public static javax.realtime.RawIntArray createRawIntArrayInstance(
  RawMemoryName type,
  long base,
  long size)
```

Creates or finds an accessor object for accessing a int array in raw memory.

type — is the required type of memory.

base — is the offset of the required int array.

size — is the length of the array.

returns an accessor object from the raw memory access.

Throws AlignmentError if the base is not on the appropriate boundary.

Throws SizeOutOfBoundsException if the short array falls in an invalid address range.

Throws MemoryTypeConflictException if base does not point to memory that matches the type served by this factory.

Throws OffsetOutOfBoundsException if the base is negative or greater than the size of the raw memory area.

```
@SCJAllowed(javax.safetycritical.annotate.Level.LEVEL_0)
@SCJRestricted({javax.safetycritical.annotate.Phase.
   INTERRUPT_SERVICE_ROUTINE}, mayAllocate = false, maySelfSuspend = false)
public static
javax.realtime.RawIntArrayRead createRawIntArrayReadInstance(
  RawMemoryName type,
  long base,
  long size)
```

Creates or finds an accessor object for read accessing a int array in raw memory.

type — is the required type of memory.

base — is the offset of the required int array.

size — is the length of the array.

returns an accessor object from the raw memory access.

Throws AlignmentError if the base is not on the appropriate boundary.

Throws SizeOutOfBoundsException if the short array falls in an invalid address range.

Throws MemoryTypeConflictException if base does not point to memory that matches the type served by this factory.

Throws OffsetOutOfBoundsException if the base is negative or greater than the size of the raw memory area.

```
@SCJAllowed(javax.safetycritical.annotate.Level.LEVEL_0)
@SCJRestricted({javax.safetycritical.annotate.Phase.
   INTERRUPT_SERVICE_ROUTINE}, mayAllocate = false, maySelfSuspend = false)
public static
javax.realtime.RawIntArrayWrite createRawIntArrayWriteInstance(
  RawMemoryName type,
  long base,
  long size)
```

Creates or finds an accessor object for write accessing a int array in raw memory.

type — is the required type of memory.

base — is the offset of the required int array.

size — is the length of the array.

returns an accessor object from the raw memory access.

Throws AlignmentError if the base is not on the appropriate boundary.

Throws SizeOutOfBoundsException if the short array falls in an invalid address range.

Throws MemoryTypeConflictException if base does not point to memory that matches the type served by this factory.

Throws OffsetOutOfBoundsException if the base is negative or greater than the size of the raw memory area.

```
@SCJAllowed(javax.safetycritical.annotate.Level.LEVEL_0)
@SCJRestricted({javax.safetycritical.annotate.Phase.
   INTERRUPT_SERVICE_ROUTINE}, mayAllocate = false, maySelfSuspend = false)
public static javax.realtime.RawInt createRawIntInstance(
  RawMemoryName type,
  long base)
```

Creates or finds an accessor object for accessing a int of raw memory.

type — is the required type of memory.

base — is the offset of the required int.

returns an accessor object from the raw memory access.

Throws AlignmentError if the base is not on the appropriate boundary.

Throws SizeOutOfBoundsException if the int falls in an invalid address range.

Throws MemoryTypeConflictException if base does not point to memory that matches the type served by this factory.

Throws OffsetOutOfBoundsException if the base is negative or greater than the size of the raw memory area.

```
@SCJAllowed(javax.safetycritical.annotate.Level.LEVEL_0)
@SCJRestricted({javax.safetycritical.annotate.Phase.
   INTERRUPT_SERVICE_ROUTINE}, mayAllocate = false, maySelfSuspend = false)
public static javax.realtime.RawIntRead createRawIntReadInstance(
  RawMemoryName type,
  long base)
```

Creates or finds an accessor object for read accessing a int of raw memory.

type — is the required type of memory.

base — is the offset of the required int.

returns an accessor object from the raw memory access.

Throws AlignmentError if the base is not on the appropriate boundary.

Throws SizeOutOfBoundsException if the int falls in an invalid address range.

Throws MemoryTypeConflictException if base does not point to memory that matches the type served by this factory.

Throws OffsetOutOfBoundsException if the base is negative or greater than the size of the raw memory area.

```
@SCJAllowed(javax.safetycritical.annotate.Level.LEVEL_0)
@SCJRestricted({javax.safetycritical.annotate.Phase.
   INTERRUPT_SERVICE_ROUTINE}, mayAllocate = false, maySelfSuspend = false)
public static javax.realtime.RawIntWrite createRawIntWriteInstance(
  RawMemoryName type,
  long base)
```

Creates or finds an accessor object for write accessing a int of raw memory.

type — is the required type of memory.

base — is the offset of the required int.

returns an accessor object from the raw memory access.

Throws AlignmentError if the base is not on the appropriate boundary.

Throws SizeOutOfBoundsException if the int falls in an invalid address range.

Throws MemoryTypeConflictException if base does not point to memory that matches the type served by this factory.

Throws OffsetOutOfBoundsException if the base is negative or greater than the size of the raw memory area.

```
@SCJAllowed(javax.safetycritical.annotate.Level.LEVEL_0)
@SCJRestricted({javax.safetycritical.annotate.Phase.
   INTERRUPT_SERVICE_ROUTINE}, mayAllocate = false, maySelfSuspend = false)
public static javax.realtime.RawLongArray createRawLongArrayInstance(
  RawMemoryName type,
  long base,
  long size)
```

Creates or finds an accessor object for accessing a long array in raw memory.

type — is the required type of memory.

base — is the offset of the required long array.

size — is the length of the array.

returns an accessor object from the raw memory access.

Throws AlignmentError if the base is not on the appropriate boundary.

Throws SizeOutOfBoundsException if the long array falls in an invalid address range.

Throws MemoryTypeConflictException if base does not point to memory that matches the type served by this factory.

Throws OffsetOutOfBoundsException if the base is negative or greater than the size of the raw memory area.

```
@SCJAllowed(javax.safetycritical.annotate.Level.LEVEL_0)
@SCJRestricted({javax.safetycritical.annotate.Phase.
   INTERRUPT_SERVICE_ROUTINE}, mayAllocate = false, maySelfSuspend = false)
public static
javax.realtime.RawLongArrayRead createRawLongArrayReadInstance(
  RawMemoryName type,
  long base,
  long size)
```

Creates or finds an accessor object for read accessing a long array in raw memory.

type — is the required type of memory.

base — is the offset of the required long array.

size — is the length of the array.

returns an accessor object from the raw memory access.

Throws AlignmentError if the base is not on the appropriate boundary.

Throws SizeOutOfBoundsException if the long array falls in an invalid address range.

Throws MemoryTypeConflictException if base does not point to memory that matches the type served by this factory.

Throws OffsetOutOfBoundsException if the base is negative or greater than the size of the raw memory area.

```
@SCJAllowed(javax.safetycritical.annotate.Level.LEVEL_0)
@SCJRestricted({javax.safetycritical.annotate.Phase.
   INTERRUPT_SERVICE_ROUTINE}, mayAllocate = false, maySelfSuspend = false)
public static
javax.realtime.RawLongArrayWrite createRawLongArrayWriteInstance(
  RawMemoryName type,
  long base,
  long size)
```

Creates or finds an accessor object for write accessing a long array in raw memory.

type — is the required type of memory.

base — is the offset of the required long array.

size — is the length of the array.

returns an accessor object from the raw memory access.

Throws AlignmentError if the base is not on the appropriate boundary.

Throws SizeOutOfBoundsException if the long array falls in an invalid address range.

Throws MemoryTypeConflictException if base does not point to memory that matches the type served by this factory.

Throws OffsetOutOfBoundsException if the base is negative or greater than the size of the raw memory area.

```
@SCJAllowed(javax.safetycritical.annotate.Level.LEVEL_0)
@SCJRestricted({javax.safetycritical.annotate.Phase.
    INTERRUPT_SERVICE_ROUTINE}, mayAllocate = false, maySelfSuspend = false)
public static javax.realtime.RawLong createRawLongInstance(
  RawMemoryName type,
  long base)
```

Creates or finds an accessor object for accessing a long in raw memory.

type — is the required type of memory.

base — is the offset of the required long.

returns an accessor object from the raw memory access.

Throws AlignmentError if the base is not on the appropriate boundary.

Throws SizeOutOfBoundsException if the long falls in an invalid address range.

Throws MemoryTypeConflictException if base does not point to memory that matches the type served by this factory.

Throws OffsetOutOfBoundsException if the base is negative or greater than the size of the raw memory area.

```
@SCJAllowed(javax.safetycritical.annotate.Level.LEVEL_0)
@SCJRestricted({javax.safetycritical.annotate.Phase.
    INTERRUPT_SERVICE_ROUTINE}, mayAllocate = false, maySelfSuspend = false)
public static javax.realtime.RawLongRead createRawLongReadInstance(
  RawMemoryName type,
  long base)
```

Creates or finds an accessor object for read accessing a long in raw memory.

type — is the required type of memory.

base — is the offset of the required long.

returns an accessor object from the raw memory access.

Throws AlignmentError if the base is not on the appropriate boundary.

Throws SizeOutOfBoundsException if the long falls in an invalid address range.

Throws MemoryTypeConflictException if base does not point to memory that matches the type served by this factory.

Throws OffsetOutOfBoundsException if the base is negative or greater than the size of the raw memory area.

```
@SCJAllowed(javax.safetycritical.annotate.Level.LEVEL_0)
@SCJRestricted({javax.safetycritical.annotate.Phase.
   INTERRUPT_SERVICE_ROUTINE}, mayAllocate = false, maySelfSuspend = false)
public static javax.realtime.RawLongWrite createRawLongWriteInstance(
  RawMemoryName type,
  long base)
```

Creates or finds an accessor object for write accessing a long in raw memory.

type — is the required type of memory.

base — is the offset of the required long.

returns an accessor object from the raw memory access.

Throws AlignmentError if the base is not on the appropriate boundary.

Throws SizeOutOfBoundsException if the long falls in an invalid address range.

Throws MemoryTypeConflictException if base does not point to memory that matches the type served by this factory.

Throws OffsetOutOfBoundsException if the base is negative or greater than the size of the raw memory area.

```
@SCJAllowed(javax.safetycritical.annotate.Level.LEVEL_0)
@SCJRestricted({javax.safetycritical.annotate.Phase.
   INTERRUPT_SERVICE_ROUTINE}, mayAllocate = false, maySelfSuspend = false)
public static javax.realtime.RawShortArray createRawShortArrayInstance(
  RawMemoryName type,
  long base,
  long size)
```

Creates or finds an accessor object for accessing a short array in raw memory.

type — is the required type of memory.

base — is the offset of the required short array.

size — is the length of the array.

returns an accessor object from the raw memory access.

Throws AlignmentError if the base is not on the appropriate boundary.

Throws SizeOutOfBoundsException if the short array falls in an invalid address range.

Throws MemoryTypeConflictException if base does not point to memory that matches the type served by this factory.

Throws OffsetOutOfBoundsException if the base is negative or greater than the size of the raw memory area.

```
@SCJAllowed(javax.safetycritical.annotate.Level.LEVEL_0)
@SCJRestricted({javax.safetycritical.annotate.Phase.
   INTERRUPT_SERVICE_ROUTINE}, mayAllocate = false, maySelfSuspend = false)
public static
javax.realtime.RawShortArrayRead createRawShortArrayReadInstance(
  RawMemoryName type,
  long base,
  long size)
```

Creates or finds an accessor object for read accessing a short array in raw memory.

type — is the required type of memory.

base — is the offset of the required short array.

size — is the length of the array.

returns an accessor object from the raw memory access.

Throws AlignmentError if the base is not on the appropriate boundary.

Throws SizeOutOfBoundsException if the short array falls in an invalid address range.

Throws MemoryTypeConflictException if base does not point to memory that matches the type served by this factory.

Throws OffsetOutOfBoundsException if the base is negative or greater than the size of the raw memory area.

```
@SCJAllowed(javax.safetycritical.annotate.Level.LEVEL_0)
@SCJRestricted({javax.safetycritical.annotate.Phase.
   INTERRUPT_SERVICE_ROUTINE}, mayAllocate = false, maySelfSuspend = false)
```

public static
javax.realtime.RawShortArrayWrite createRawShortArrayWriteInstance(
  RawMemoryName type,
  long base,
  long size)

Creates or finds an accessor object for write accessing a short array in raw memory.

type — is the required type of memory.

base — is the offset of the required short array.

size — is the length of the array.

returns an accessor object from the raw memory access.

Throws AlignmentError if the base is not on the appropriate boundary.

Throws SizeOutOfBoundsException if the short array falls in an invalid address range.

Throws MemoryTypeConflictException if base does not point to memory that matches the type served by this factory.

Throws OffsetOutOfBoundsException if the base is negative or greater than the size of the raw memory area.

@SCJAllowed(javax.safetycritical.annotate.Level.LEVEL_0)
@SCJRestricted({javax.safetycritical.annotate.Phase.
   INTERRUPT_SERVICE_ROUTINE}, mayAllocate = false, maySelfSuspend = false)
public static javax.realtime.RawShort createRawShortInstance(
  RawMemoryName type,
  long base)

Creates or finds an accessor object for accessing a short of raw memory.

type — is the required type of memory.

base — is the offset of the required short.

returns an accessor object from the raw memory access.

Throws AlignmentError if the base is not on the appropriate boundary.

Throws SizeOutOfBoundsException if the short falls in an invalid address range.

Throws MemoryTypeConflictException if base does not point to memory that matches the type served by this factory.

Throws OffsetOutOfBoundsException if the base is negative or greater than the size of the raw memory area.

```
@SCJAllowed(javax.safetycritical.annotate.Level.LEVEL_0)
@SCJRestricted({javax.safetycritical.annotate.Phase.
   INTERRUPT_SERVICE_ROUTINE}, mayAllocate = false, maySelfSuspend = false)
public static javax.realtime.RawShortRead createRawShortReadInstance(
  RawMemoryName type,
  long base)
```

Creates or finds an accessor object for read accessing a short of raw memory.

type — is the required type of memory.

base — is the offset of the required short.

returns an accessor object from the raw memory access.

Throws AlignmentError if the base is not on the appropriate boundary.

Throws SizeOutOfBoundsException if the short falls in an invalid address range.

Throws MemoryTypeConflictException if base does not point to memory that matches the type served by this factory.

Throws OffsetOutOfBoundsException if the base is negative or greater than the size of the raw memory area.

```
@SCJAllowed(javax.safetycritical.annotate.Level.LEVEL_0)
@SCJRestricted({javax.safetycritical.annotate.Phase.
   INTERRUPT_SERVICE_ROUTINE}, mayAllocate = false, maySelfSuspend = false)
public static javax.realtime.RawShortWrite createRawShortWriteInstance(
  RawMemoryName type,
  long base)
```

Creates or finds an accessor object for write accessing a short of raw memory.

type — is the required type of memory.

base — is the offset of the required short.

returns an accessor object from the raw memory access.

Throws AlignmentError if the base is not on the appropriate boundary.

Throws SizeOutOfBoundsException if the short falls in an invalid address range.

Throws MemoryTypeConflictException if base does not point to memory that matches the type served by this factory.

Throws OffsetOutOfBoundsException if the base is negative or greater than the size of the raw memory area.

@SCJAllowed(javax.safetycritical.annotate.Level.LEVEL_0)
@SCJRestricted(mayAllocate = false, maySelfSuspend = false)
public static void registerAccessFactory(
  RawIntegralAccessFactory factory)

Registers a factory for accessing raw memory.

factory — is the factory being registered.

Throws java.lang.IllegalArgumentException if factory is null or its name is served by a factory that has already been registered.

D.2.40 CLASS **RealtimeThread**

@SCJAllowed(javax.safetycritical.annotate.Level.LEVEL_1)
public class RealtimeThread
  implements javax.realtime.Schedulable
  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 getCurrentMemoryArea method can be used at Level 1, hence the class is visible at Level 1.

Methods

@SCJAllowed(javax.safetycritical.annotate.Level.LEVEL_1)
@SCJRestricted(maySelfSuspend = false, mayAllocate = false)
public static javax.realtime.MemoryArea getCurrentMemoryArea( )

Allocates no memory. The returned object may reside in scoped memory, within a scope that encloses the current execution context.

returns a reference to the current allocation context.

@SCJAllowed(javax.safetycritical.annotate.Level.LEVEL_2)
@SCJRestricted(maySelfSuspend = false, mayAllocate = false)
public javax.realtime.MemoryArea getMemoryArea( )

Allocates no memory. Does not allow this to escape local variables. The returned object may reside in scoped memory, within a scope that encloses this.

returns a reference to the initial allocation context represented by this.

@SCJAllowed(javax.safetycritical.annotate.Level.LEVEL_2)
@SCJRestricted(maySelfSuspend = true)
public static void sleep(HighResolutionTime time)
  throws java.lang.InterruptedException

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

Throws java.lang.IllegalArgumentException if time is based on a user-defined clock that does not drive events.

D.2.41 CLASS **RegistrationException**

@SCJAllowed
public class RegistrationException **extends** java.lang.RuntimeException

Kelvin added this on 2/8/11 because it is required by InterruptServiceRoutine. Did I get the right superclass?

Constructors

@SCJAllowed @SCJRestricted(maySelfSuspend = false)
public RegistrationException( )

@SCJAllowed @SCJRestricted(maySelfSuspend = false)
public RegistrationException(String description)

Kelvin wonders why this should be declared to allocate in immortal. This code was copied from MemoryScopeException. Why should that allocate in immortal?

Memory behavior: This constructor may allocate objects within the ImmortalMemory MemoryArea.

D.2.42 CLASS **RelativeTime**

@SCJAllowed
public class RelativeTime **extends** javax.realtime.HighResolutionTime

An object that represents a time interval milliseconds/10$\hat{3}$ + nanoseconds/10$\hat{9}$ seconds long that is divided into subintervals by some frequency. This is generally used in periodic events, threads, and feasibility analysis to specify periods where there is a basic period that must be adhered to strictly (the interval), but within that interval the periodic events are supposed to happen frequency times, as uniformly spaced as possible, but clock and scheduling jitter is moderately acceptable.

Constructors

@SCJAllowed @SCJRestricted(maySelfSuspend = false)
public RelativeTime( )

Equivalent to new RelativeTime(0,0).

@SCJAllowed @SCJRestricted(maySelfSuspend = false)
public RelativeTime(**long** ms, **int** ns)

Construct a RelativeTime object representing an interval based on the parameter millis plus the parameter nanos. Todo: say which clock this RelativeTime is associated with.

ms — The desired value for the millisecond component of this. The actual value is the result of parameter normalization.

ns — The desired value for the nanosecond component of this. The actual value is the result of parameter normalization.

@SCJAllowed @SCJRestricted(maySelfSuspend = false)
public RelativeTime(Clock clock)

Equivalent to new RelativeTime(0,0,clock).

clock — The clock providing the association for the newly constructed object.

@SCJAllowed @SCJRestricted(maySelfSuspend = false)
public RelativeTime(**long** ms, **int** ns, Clock clock)

Construct a RelativeTime object representing an interval based on the parameter millis plus the parameter nanos.

ms — The desired value for the millisecond component of this. The actual value is the result of parameter normalization.

ns — The desired value for the nanosecond component of this. The actual value is the result of parameter normalization.

clock — The clock providing the association for the newly constructed object.

@SCJAllowed @SCJRestricted(maySelfSuspend = false)
public RelativeTime(RelativeTime time)

Make a new RelativeTime object from the given RelativeTime object.

time — The RelativeTime object which is the source for the copy.

Methods

@SCJAllowed @SCJRestricted(maySelfSuspend = false)
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.

time — The time to add to this.

returns A new RelativeTime object whose time is the normalization of this plus millis and nanos.

Memory behavior: This constructor may allocate objects within the currently active MemoryArea.

@SCJAllowed @SCJRestricted(maySelfSuspend = false)
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.

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 result of the normalization of this plus the RelativeTime parameter time in dest if dest is not null, otherwise the result is returned in a newly allocated object.

```
@SCJAllowed @SCJRestricted(maySelfSuspend = false)
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.

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 result of the normalization of this plus millis and nanos in dest if dest is not null, otherwise the result is returned in a newly allocated object.

```
@SCJAllowed @SCJRestricted(maySelfSuspend = false)
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.

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.

Memory behavior: This constructor may allocate objects within the currently active MemoryArea.

```
@SCJAllowed @SCJRestricted(maySelfSuspend = false)
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.

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 result of the normalization of this minus the RelativeTime parameter time in dest if dest is not null, otherwise the result is returned in a newly allocated object.

@SCJAllowed @SCJRestricted(maySelfSuspend = false)
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.

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.

Memory behavior: This constructor may allocate objects within the currently active MemoryArea.

D.2.43 CLASS **ReleaseParameters**

@SCJAllowed
public abstract class ReleaseParameters
implements java.lang.Cloneable
extends java.lang.Object

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

Constructors

@SCJAllowed
protected ReleaseParameters( )

Construct a ReleaseParameters object which has no deadline checking facility. There is no default for the deadline in this class. The default is set by the subclasses.

@SCJAllowed(javax.safetycritical.annotate.Level.LEVEL_1)
protected ReleaseParameters(RelativeTime deadline,
  AsyncEventHandler missHandler)

Construct an object which has deadline checking facility.

deadline — is a deadline to be checked.

missHandler — is the AsynchronousEventHandler to be released when the deadline miss has been detected.

Methods

@SCJAllowed
public java.lang.Object clone( )

Create a clone of this ReleaseParameters object.

D.2.44 CLASS Scheduler

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

The RTSJ supports generic on-line feasibility analysis via the Scheduler class. SCJ supports off-line analysis; hence most of the methods in this class are omitted. Only the static method getCurrentSO is provided.

Methods

@SCJAllowed
public static javax.realtime.Schedulable getCurrentSO( )

returns the current asynchronous event handler or real-time thread of the caller.

D.2.45 CLASS SchedulingParameters

@SCJAllowed
public abstract class SchedulingParameters
  implements java.lang.Cloneable
  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.

D.2.46 CLASS ScopedCycleException

@SCJAllowed
public class ScopedCycleException
  implements java.io.Serializable
  extends java.lang.RuntimeException

Constructors

@SCJAllowed @SCJRestricted(maySelfSuspend = false)
public ScopedCycleException( )

Memory behavior: This constructor may allocate objects within the currently active MemoryArea.

@SCJAllowed @SCJRestricted(maySelfSuspend = false)
public ScopedCycleException(String description)

Memory behavior: This constructor may allocate objects within the currently active MemoryArea.

This constructor requires that the "msg" argument reside in a scope that encloses the scope of the "this" argument.

D.2.47 CLASS ScopedMemory

@SCJAllowed
public abstract class ScopedMemory
  implements javax.realtime.ScopedAllocationContext
  extends javax.realtime.MemoryArea

Scoped memory implements the scoped allocation context. The only visible method is for resizing a scoped memory area.

D.2.48 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.

SizeEstimator is a ceiling on the amount of memory that is consumed when the reserved objects are created.

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 (if, for example, the object and its monitor are separate), the size estimate shall include the sum of the sizes of all the 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.

Constructors

@SCJAllowed
@SCJRestricted(maySelfSuspend = false,
{javax.safetycritical.annotate.Phase.ALL})
public SizeEstimator( )

Creates a new SizeEstimator object in the current allocation context.

Methods

@SCJAllowed
@SCJRestricted(maySelfSuspend = false,
{javax.safetycritical.annotate.Phase.ALL})
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
@SCJRestricted(maySelfSuspend = false,
{javax.safetycritical.annotate.Phase.ALL})
public void reserve(SizeEstimator size, **int** num)

Take into account additional number instances of SizeEstimator when estimating the size.

size — is the given instance of SizeEstimator.

num — is the number of times to reserve the size denoted by size.

Throws IllegalArgumentException if size is null.

```
@SCJAllowed
@SCJRestricted(maySelfSuspend = false,
   {javax.safetycritical.annotate.Phase.ALL})
public void reserve(SizeEstimator size)
```

Take into account an additional instance of SizeEstimator size when estimating the size.

size — is the given instance of SizeEstimator.

Throws IllegalArgumentException if size is null.

```
@SCJAllowed
@SCJRestricted(maySelfSuspend = false,
   {javax.safetycritical.annotate.Phase.ALL})
public void reserve(Class<> clazz, int num)
```

Take into account additional num instances of Class clazz when estimating the size.

clazz — is the class to take into account.

num — is the number of instances of clazz to estimate.

Throws IllegalArgumentException if clazz is null.

```
@SCJAllowed
@SCJRestricted(maySelfSuspend = false,
   {javax.safetycritical.annotate.Phase.ALL})
public void reserveArray(int length, Class<> type)
```

Take into account an additional instance of an array of length primitive values. Class values for the primitive types are available from the corresponding class types; e.g., Byte.TYPE, Integer.TYPE, and Short.TYPE. The reservation will 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
@SCJRestricted(maySelfSuspend = false,
   {javax.safetycritical.annotate.Phase.ALL})
public void reserveArray(int length)
```

Take into account an additional instance of an array of length reference values.

length — is the number of entries in the array.

D.2.49 CLASS **Test**

```
@SCJAllowed(javax.safetycritical.annotate.Level.LEVEL_1)
public abstract class Test extends java.lang.Object
```

The base level class used to support first level handling. Application-defined subclasses override the handle method to define the first-level service routine

Constructors

```
@SCJAllowed(javax.safetycritical.annotate.Level.LEVEL_1)
public Test(String name)
```

Creates an interrupt signal with the given name and associated with a given interrupt.

name — is a system dependent designator for the interrupt

Methods

```
@SCJAllowed(javax.safetycritical.annotate.Level.LEVEL_1)
protected abstract int test(int param1, int param2)
  throws java.lang.IllegalArgumentException
```

A test method with parameters and return values. Throws **javax.realtime-.IllegalArgumentException** when error.

param1 — is a name.

param2 — is a name.

returns integer vale.

Throws IllegalArgumentException when error. @code while (has registeredHap-pening) doit();

D.2.50 CLASS ThrowBoundaryError

@SCJAllowed
public class ThrowBoundaryError
  implements java.io.Serializable
  extends java.lang.Error

Appendix E

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*

E.1 Interfaces

E.1.1 INTERFACE ManagedSchedulable

@SCJAllowed
public interface ManagedSchedulable
  implements javax.realtime.Schedulable

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 applications classes.

Methods

@SCJAllowed(javax.safetycritical.annotate.Level.SUPPORT)
@SCJRestricted({javax.safetycritical.annotate.Phase.CLEANUP})
public void cleanUp( )

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

E.1.2 INTERFACE Safelet

@SCJAllowed
public interface Safelet<MissionLevel **extends** Mission>

A safety-critical application consists of one or more missions, executed concurrently or in sequence. Every safety-critical application is represented by an implementation of Safelet which identifies the outer-most MissionSequencer. This outer-most MissionSequencer takes responsibility for running 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.

For the MissionSequencer returned from getSequencer, the SCJ infrastructure arranges for an independent thread to begin executing the code for that sequencer and then waits for that thread to terminate its execution.

Methods

@SCJAllowed(javax.safetycritical.annotate.Level.SUPPORT)
@SCJRestricted({javax.safetycritical.annotate.Phase.INITIALIZATION})
public
javax.safetycritical.MissionSequencer<MissionLevel> 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)
public long immortalMemorySize( )

returns the amount of additional immortal memory that must be available for the immortal memory allocations to be performed by this application. If the amount of memory remaining in immortal memory is less than this requested size, the infrastructure halts execution of the application upon return from this method.

@SCJAllowed(javax.safetycritical.annotate.Level.SUPPORT)
@SCJRestricted({javax.safetycritical.annotate.Phase.INITIALIZATION})
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 that are in immortal memory. initializeApplication shall be invoked after immortalMemorySize, and before getSequencer.

E.1.3 INTERFACE **Schedulable**

@SCJAllowed
public interface Schedulable **implements** java.lang.Runnable

Methods

@SCJAllowed(javax.safetycritical.annotate.Level.LEVEL_2)
@SCJRestricted(maySelfSuspend = false)
public
javax.safetycritical.StorageParameters getThreadConfigurationParameters( )

Does not allocate memory. Does not allow this to escape local variables. Returns an object that resides in the corresponding thread's MissionMemory scope.

E.2 Classes

E.2.1 CLASS **AperiodicEvent**

@SCJAllowed(javax.safetycritical.annotate.Level.LEVEL_1)
public class AperiodicEvent **extends** javax.realtime.AsyncEvent

A class of events that enables application code to release AperiodicEventHandlers. The event is software-triggered by the application or the infrastructure.

Constructors

@SCJAllowed(javax.safetycritical.annotate.Level.LEVEL_1)
@SCJRestricted({javax.safetycritical.annotate.Phase.INITIALIZATION})
public AperiodicEvent( )

Construct an aperiodic event.

Memory behavior: Does not allocate memory. Does not allow this to escape the local variables.

E.2.2 CLASS **AperiodicEventHandler**

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

extends javax.safetycritical.ManagedEventHandler

This class permits the automatic execution of code that is bound to an aperiodic event. It is abstract. Concrete subclasses must implement the handleAsync-Event method and may override the default cleanup method.

Note, there is no programmer access to the RTSJ fireCount mechanisms, so the associated methods are missing.

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)
@SCJRestricted({javax.safetycritical.annotate.Phase.INITIALIZATION})
public AperiodicEventHandler(PriorityParameters priority,
AperiodicParameters release,
StorageParameters storage)

Constructs an aperiodic event handler that can be explicitly released.

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

release — specifies the release parameters for this aperiodic event handler; it must not be null.
storage — specifies the StorageParameters for this aperiodic event handler
Throws IllegalArgumentException IllegalArgumentException if priority, release or event is null.

Memory behavior: Does not perform memory allocation. Does not allow this to escape local scope. Builds links from this to priority and parameters, so those two arguments must reside in scopes that enclose this. Builds a link from this to the event, so the event must reside in memory that encloses this.

Methods

@SCJAllowed
public final void release( )

Release this aperiodic event handler

E.2.3 CLASS **AperiodicLongEvent**

@SCJAllowed(javax.safetycritical.annotate.Level.LEVEL_1)
public class AperiodicLongEvent **extends** javax.realtime.AsyncLongEvent

A class of events that enables code to release AperiodicEventHandlers and pass a parameter. The event is software-triggered.

This class differs from AperiodicEvent in that when it is fired, a long integer is provided for use by the released event handler(s).

Constructors

@SCJAllowed(javax.safetycritical.annotate.Level.LEVEL_1)
@SCJRestricted({javax.safetycritical.annotate.Phase.INITIALIZATION})
public AperiodicLongEvent( )

Constructs an aperiodic event that allows a long parameter to be passed when it is fired.

Memory behavior: Does not allocate memory. Does not allow this to escape the local variables.

Memory behavior: This constructor may allocate objects within the same MemoryArea that holds the implicit this argument.

This constructor requires that the "handler" argument reside in a scope that encloses the scope of the "this" argument.

E.2.4 CLASS **AperiodicLongEventHandler**

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

extends javax.safetycritical.ManagedLongEventHandler

This class permits the automatic execution of code that is bound to an aperiodic event. It is abstract. Concrete subclasses must implement the handleAsyncEvent method and may override the default cleanup method.

Note, there is no programmer access to the RTSJ fireCount mechanisms, so the associated methods are missing.

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)
@SCJRestricted({javax.safetycritical.annotate.Phase.INITIALIZATION})
public AperiodicLongEventHandler(PriorityParameters priority,
AperiodicParameters release,
StorageParameters storage)

Constructs an aperiodic event handler that can be released.

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

release — specifies the release parameters for this aperiodic event handler; it must not be null.

storage — specifies the storage parameters for the periodic event handler. It must not be null.

Throws IllegalArgumentException IllegalArgumentException if priority, release or event is null.

Memory behavior: Does not perform memory allocation. Does not allow this to escape local scope. Builds links from this to priority and parameters, so those two arguments must reside in scopes that enclose this. Builds a link from this to event, so event must reside in memory that encloses this.

Methods

@SCJAllowed
public final void release(**long** data)

Release this aperiodic event handler

E.2.5 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
public CyclicExecutive( )

Construct a CyclicExecutive object.

Methods

@SCJAllowed(javax.safetycritical.annotate.Level.SUPPORT)
public 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 override the getSchedule 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 MissionMemory area shall be the active allocation context. The value returned from getSchedule must reside in the current mission's MissionMemory 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.

Memory behavior: This constructor requires that the "handlers" argument reside in a scope that encloses the scope of the "this" argument.

E.2.6 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
public 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.
Throws IllegalStateException if invoked in a Level 1 a Level 2 application.

Memory behavior: This constructor may allocate objects within the same MemoryArea that holds the implicit this argument.
This constructor requires that the "frames" argument reside in a scope that encloses the scope of the "this" argument.

E.2.7 CLASS Frame

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

Constructors

@SCJAllowed
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. Under normal circumstances, this Frame object is instantiated within the MissionMemory area that corresponds to the Level0Mission that is to be scheduled.

Within each execution frame of the CyclicSchedule, the PeriodicEventHandler objects represented by the handlers array will be fired in same order as they appear within this array. Normally, PeriodicEventHandlers are sorted into decreasing priority order prior to invoking this constructor.

Memory behavior: This constructor may allocate objects within the same MemoryArea that holds the implicit this argument.
This constructor requires that the "duration" argument reside in a scope that encloses the scope of the "this" argument. This constructor requires that the "handlers" argument reside in a scope that encloses the scope of the "this" argument.

E.2.8 CLASS InterruptHandler

@SCJAllowed(javax.safetycritical.annotate.Level.LEVEL_1)
public abstract class InterruptHandler **extends** java.lang.Object

Constructors

@SCJAllowed(javax.safetycritical.annotate.Level.LEVEL_1)
public InterruptHandler(**int** InterruptID)

Create and register an interrupt handler. Can only be called during the initialization phase of a mission. The interrupt is automatically enabled. The ceiling of the objects is set to the hardware priority of the interrupt. It is assumed that the associated MissionManager will unregister the interrupt handler on mission termination.

Throws IllegalArgument when InterruptId is unsupported

Throws IllegalStateException when a handler is already registered or if called outside the initialization phase.

Methods

@SCJAllowed(javax.safetycritical.annotate.Level.LEVEL_1)
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 IllegalArgument if unsupported InterruptId

@SCJAllowed(javax.safetycritical.annotate.Level.LEVEL_1)
public synchronized void handleInterrupt( )

Override this method to provide the first level interrupt handler.

E.2.9 CLASS **LinearMissionSequencer**

@SCJAllowed
public class LinearMissionSequencer<SpecificMission>

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 <SpecificMission> 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
@SCJRestricted({javax.safetycritical.annotate.Phase.INITIALIZATION},
  maySelfSuspend = false)
public LinearMissionSequencer(PriorityParameters priority,
  StorageParameters storage,

```
  boolean repeat,
  SpecificMission m)
  throws java.lang.IllegalArgumentException,
      java.lang.IllegalStateException
```

Construct a LinearMissionSequencer object to oversee execution of the 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.
repeat — When repeat is true, the specified mission shall be repeated indefinitely.
m — The single mission that runs under the oversight of this LinearMissionSequencer.
Throws IllegalArgumentException if any of the arguments equals null.

Memory behavior: This constructor requires that the "priority" argument reside in a scope that encloses the scope of the "this" argument. This constructor requires that the "storage" argument reside in a scope that encloses the scope of the "this" argument. This constructor requires that the "m" argument reside in a scope that encloses the scope of the "this" argument.

```
@SCJAllowed
@SCJRestricted({javax.safetycritical.annotate.Phase.INITIALIZATION},
    maySelfSuspend = false)
public LinearMissionSequencer(PriorityParameters priority,
  StorageParameters storage,
  boolean repeat,
  SpecificMission m,
  String name)
  throws java.lang.IllegalArgumentException,
      java.lang.IllegalStateException
```

Construct a LinearMissionSequencer object to oversee execution of the 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.
repeat — When repeat is true, the specified mission shall be repeated indefinitely.
m — The single mission that runs under the oversight of this LinearMissionSequencer.
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.

Memory behavior: This constructor requires that the "priority" argument reside in a scope that encloses the scope of the "this" argument. This constructor requires that the "storage" argument reside in a scope that encloses the scope of the "this" argument. This constructor requires that the "m" argument reside in a scope that encloses the scope of the "this" argument. This constructor requires that the "name" argument reside in a scope that encloses the scope of the "this" argument.

```
@SCJAllowed
@SCJRestricted({javax.safetycritical.annotate.Phase.INITIALIZATION},
   maySelfSuspend = false)
public LinearMissionSequencer(PriorityParameters priority,
  StorageParameters storage,
  SpecificMission [] missions,
  boolean repeat)
  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.

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. It is required 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.

Throws IllegalArgumentException if any of the arguments equals null.

Memory behavior: This constructor requires that the "priority" argument reside in a scope that encloses the scope of the "this" argument. This constructor requires that the "storage" argument reside in a scope that encloses the scope of the "this" argument.

```
@SCJAllowed
```

```
@SCJRestricted({javax.safetycritical.annotate.Phase.INITIALIZATION},
   maySelfSuspend = false)
public LinearMissionSequencer(PriorityParameters priority,
  StorageParameters storage,
  SpecificMission [] 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.

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.

Memory behavior: This constructor requires that the "priority" argument reside in a scope that encloses the scope of the "this" argument. This constructor requires that the "storage" argument reside in a scope that encloses the scope of the "this" argument. This constructor requires that the "name" argument reside in a scope that encloses the scope of the "this" argument.

Methods

```
@SCJAllowed(javax.safetycritical.annotate.Level.SUPPORT)
@SCJRestricted({javax.safetycritical.annotate.Phase.INITIALIZATION},
   maySelfSuspend = false)
@Override
protected final SpecificMission 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 Also: javax.safetycritical.MissionSequencer.getNextMission()

E.2.10 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.

Note that the values in parameter 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.

Methods

@SCJAllowed(javax.safetycritical.annotate.Level.SUPPORT)
@SCJRestricted({javax.safetycritical.annotate.Phase.CLEANUP})
public void cleanUp( )

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

MissionMemory is the current allocation context on entry into this method. When the cleanUp method is called, a private memory area shall be provided for its use, and shall be the current memory area. If desired, the cleanUp method may introduce a new PrivateMemory area. The memory allocated to ManagedSchedulables shall be available to be reclaimed when each Mission's cleanUp method returns.

@SCJAllowed
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.

E.2.11 CLASS **ManagedInterruptServiceRoutine**

@SCJAllowed(javax.safetycritical.annotate.Level.LEVEL_1)
public abstract class ManagedInterruptServiceRoutine

extends javax.realtime.InterruptServiceRoutine

Constructors

@SCJAllowed(javax.safetycritical.annotate.Level.LEVEL_1)
public ManagedInterruptServiceRoutine(**long** sizes)

Creates an interrupt service routine with the given name and associated with a given interrupt.

sizes — defines the memory space required by the handle method.

initialMemoryAreaSize — is the size of a private memory area which acts as the initial allocation context for the handle method. A size of 0 indicates that any use of the new operator within the initial allocation context will result in an OutOfMemoryException being thrown.

Methods

@SCJAllowed(javax.safetycritical.annotate.Level.LEVEL_1)
@Override
@SCJRestricted({javax.safetycritical.annotate.Phase.INITIALIZATION})
public final void register(**int** interrupt)
  throws javax.realtime.RegistrationException

Equivalent to register(interrupt, highestInterruptCeilingPriority).

@SCJAllowed(javax.safetycritical.annotate.Level.LEVEL_1)
@SCJRestricted({javax.safetycritical.annotate.Phase.INITIALIZATION})
public final void register(**int** interrupt, **int** ceiling)
  throws javax.realtime.RegistrationException

Registers the ISR for the given interrupt with the current mission, 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 required ceiling is not as high or higher than this interrupt priority.

@SCJAllowed(javax.safetycritical.annotate.Level.LEVEL_1)
@SCJRestricted({javax.safetycritical.annotate.Phase.
  INTERRUPT_SERVICE_ROUTINE})
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)
@SCJRestricted({javax.safetycritical.annotate.Phase.CLEANUP})
public final void unregister( )

Unregisters the ISR with the current mission.

E.2.12 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 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. They set up the initial memory area of each managed handler to be a private memory that is entered before a call to handleAsyncEvent and that is left on return. The size of the private memory area allocated is the maximum available to the infrastructure for this handler.

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.

This class differs from ManagedEventHandler in that when it is fired, a long integer is provided for use by the released event handler(s).

Constructors

@SCJAllowed
@SCJRestricted({javax.safetycritical.annotate.Phase.INITIALIZATION})
public ManagedLongEventHandler(PriorityParameters priority,
ReleaseParameters release,
StorageParameters storage,
String name)

Constructs an event handler.

Does not perform memory allocation. Does not allow this to escape local scope. Builds links from this to priority, parameters, and name so those three arguments must reside in scopes that enclose this.

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 and period. 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. This argument must not be null.

storage — specifies the non-null maximum memory demands for this event handler.

Throws IllegalArgumentException IllegalArgumentException if priority, release or memory parameters are null.

Methods

```
@Override
@SCJAllowed(javax.safetycritical.annotate.Level.SUPPORT)
@SCJRestricted({javax.safetycritical.annotate.Phase.CLEANUP})
public void cleanUp( )
```

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

MissionMemory is the current allocation context on entry into this method.

```
@SCJAllowed
public java.lang.String getName( )
```

returns a string name for this handler, including its priority and its level.Registers this event handler with the current mission.

E.2.13 CLASS ManagedMemory

```
@SCJAllowed
public abstract class ManagedMemory extends javax.realtime.LTMemory
```

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. Applications shall not directly extend this class.

Methods

```
@SCJAllowed
public static void enterPrivateMemory(long size, Runnable logic)
  throws java.lang.IllegalStateException
```

This method causes the calling schedulable object to execute the logic in a nested private memory area. If a private memory does not exist, one is created create with the specified size; otherwise, its size is set. Then the private memory area is cleared and entered.

The private memory object representing the inner scope memory area may be reused on subsequent calls to enterPrivateMemory during the lifetime of the current memory area.

size — is the size in bytes of the private memory.
logic — is the code to be executed in the private memory.

Throws IllegalStateException if the currently running thread is forbidden from entering a PrivateMemory area, such as when the current thread is executing within Mission.initialize or Safelet.getSequencer,

@SCJAllowed
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 that is allocated in the memory area that is entered.
logic — is the code to be executed in the entered memory area.

@SCJAllowed
public static void executeInOuterArea(Runnable logic)

Change the allocation context to the immediate outer memory area and invoke the run method if the Runnable.

logic — is the code to be executed in the entered memory area.

@SCJAllowed
public long getRemainingBackingStore( )

This method can be used to manage the use of backing store by any SCJ memory area.

returns the size of the remaining memory available to the current ManagedMemory area.

E.2.14 CLASS **ManagedThread**

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

This class enables a mission to keep track of all the no-heap realtime threads that are created during the initialization phase. It also sets up the initial memory area for the thread to be a private memory, whose size is the maximum that the infrastructure can assign to this thread.

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

```
@SCJAllowed(javax.safetycritical.annotate.Level.LEVEL_2)
@SCJRestricted({javax.safetycritical.annotate.Phase.INITIALIZATION})
public ManagedThread(SchedulingParameters priority,
  StorageParameters storage)
```

Constructs a thread that is managed by the enclosing mission.

Does not allow this to escape local variables. Creates a link from the constructed object to the scheduling, storage, and logic parameters. Thus, all of these parameters must reside in a scope that encloses this.

The priority represented by scheduling parameter is consulted only once, at construction time. If scheduling.getPriority() returns different values at different times, only the initial value is honored.

priority — specifies the priority parameters for this managed thread; it must not be null.
storage — specifies the storage parameters for this thread. May not be null.
Throws IllegalArgumentException if priority or storage is null.

Memory behavior: This constructor requires that the "priority" argument reside in a scope that encloses the scope of the "this" argument. This constructor requires that the "storage" argument reside in a scope that encloses the scope of the "this" argument.

```
@SCJAllowed(javax.safetycritical.annotate.Level.LEVEL_2)
@SCJRestricted({javax.safetycritical.annotate.Phase.INITIALIZATION})
public ManagedThread(PriorityParameters priority,
  StorageParameters storage,
  Runnable logic)
```

Creates a thread that is managed by the enclosing mission.

Does not allow this to escape local variables. Creates a link from the constructed object to the priority, memory, and logic parameters . Thus, all of these parameters must reside in a scope that encloses this.

The priority represented by priority parameter is consulted only once, at construction time. If priority.getPriority() returns different values at different times, only the initial value is honored.

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.

Memory behavior: This constructor requires that the "priority" argument reside in a scope that encloses the scope of the "this" argument. This constructor requires that the "mem_info" argument reside in a scope that encloses the scope of the "this" argument. This constructor requires that the "logic" argument reside in a scope that encloses the scope of the "this" argument.

Methods

@Override
@SCJAllowed(javax.safetycritical.annotate.Level.LEVEL_2)
@SCJRestricted({})
public void cleanUp( )

Execute any clean up code associated with this managed thread. This method is called by the infrastructure, so that it is not callable from any phase. In fact, it encapsulates the entire cleanup phase.

@Override
@SCJAllowed(javax.safetycritical.annotate.Level.LEVEL_0)
@SCJRestricted({javax.safetycritical.annotate.Phase.INITIALIZATION})
public final void register( )

Register this managed thread. Note: method made Level 0 since it overrides ManagedSchedulable.register() method which is also Level 0. Note, however, that this does not make the method visible at Level 0 since the enclosing class is Level 2.

E.2.15 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
public Mission( )

> 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 he 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.

Memory behavior: This constructor may allocate objects within the same MemoryArea that holds the implicit this argument.

Methods

@SCJAllowed(javax.safetycritical.annotate.Level.SUPPORT)
protected void cleanUp( )

> Method to clean data structures and machine state upon termination of this Mission's execute phase. Infrastructure code running in the controlling MissionSequencer's bound thread invokes cleanUp after all ManagedSchedulables associated with this Mission have terminated, but before control leaves the corresponding MissionMemory area. The default implementation of cleanUp does nothing.

@SCJAllowed
public static javax.safetycritical.Mission getCurrentMission( )

> Obtain the current mission.

returns the instance of the Mission to which the currently executing ManagedSchedulable corresponds. The current Mission is known from the moment when initialize has been invoked and continues to be known until the mission's last cleanUp method has been completed. Otherwise, returns null.

@SCJAllowed
public javax.safetycritical.MissionSequencer> getSequencer( )

returns the MissionSequencer that is overseeing execution of this mission.

@SCJAllowed(javax.safetycritical.annotate.Level.SUPPORT)
protected abstract void initialize( )

Perform initialization of this Mission. Infrastructure calls initialize after the Mission has been instantiated and the MissionMemory has been resized to match the size returned from Mission.missionMemorySize. Upon entry into the initialize method, the current allocation context is the MissionMemory area dedicated to this particular Mission.

The default implementation of initialize does nothing.

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 under the direction of the SCJ infrastructure. 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 certain mission-level data structures. Note that objects shared between ManagedSchedulables typically reside within the corresponding MissionMemory scope, but may alternatively reside in outer-nested MissionMemory 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 (the value returned from Mission.getCurrentMission), coercing the reference to the known Mission subclass, and accessing the fields or methods of this subclass that represent the shared data objects.

@SCJAllowed(javax.safetycritical.annotate.Level.SUPPORT)
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 MissionMemory area. When this method receives control, the MissionMemory 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 PrivateMemory areas. After this method returns, the SCJ infrastructure shall shrink the MissionMemory 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 PrivateMemory area within this method will result in an OutOfMemoryError exception.

@SCJAllowed
public final void requestTermination( )

This method provides a standard interface for requesting termination of a mission. Once this method is called during Mission execution, subsequent invocations of terminationPending shall return true, shall invoke this object's terminationHook method, and shall invoke requestSequenceTermination on each inner-nested MissionSequencer object that is registered for execution within this mission. Additionally, this method has the effect of arranging to (1) disable all periodic event handlers associated with this Mission so that they will experience no further firings, (2) disable all AperiodicEventHandlers so that no further firings will be honored, (3) clear the pending event (if any) for each event handler 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 invoking 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 after 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.

@SCJAllowed(javax.safetycritical.annotate.Level.SUPPORT)
protected void terminationHook( )

This method shall be invoked by requestTermination. Application-specific subclasses of Mission may override the terminationHook method to supply application-specific mission shutdown code.

E.2.16 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.

E.2.17 CLASS MissionSequencer

@SCJAllowed
public abstract class MissionSequencer<SpecificMission **extends** Mission>

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 is bound to an event handling thread. The bound thread's execution priority and memory budget are specified by constructor parameters.

This MissionSequencer executes vendor-supplied infrastructure code which invokes user-defined implementations of MissionSequencer. getNextMission, Mission.initialize, and Mission.cleanUp. During execution of an inner-nested mission, the MissionSequencer's thread remains blocked waiting for the mission to terminate. An invocation of MissionSequencer.request SequenceTermination will unblock this waiting thread so that it can perform an invocation of the running mission's requestTermination method if the mission is still running and its termination has not already been requested.

Note that if a MissionSequencer object is preallocated by the application, it must be allocated in the same scope as its corresponding Mission.

Constructors

@SCJAllowed
@SCJRestricted({javax.safetycritical.annotate.Phase.INITIALIZATION})
public MissionSequencer(PriorityParameters priority,
StorageParameters storage,
String name)
throws java.lang.IllegalStateException

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

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.

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

Throws IllegalStateException if invoked at an inappropriate time. The only appropriate times for instantiation of a new MissionSequencer are (a) during execution of Safelet.getSequencer by SCJ infrastructure during startup of an SCJ application, and (b) during execution of Mission.initialize by SCJ infrastructure during initialization of a new mission in a Level 2 configuration of the SCJ run-time environment.

Note that the static checker for SCJ forbids instantiation of MissionSequencer objects outside of mission initialization, but it does not prevent Mission.initialize in a Level 1 application from attempting to instantiate a MissionSequencer.

Memory behavior: This constructor requires that the "priority" argument reside in a scope that encloses the scope of the "this" argument. This constructor requires that the "storage" argument reside in a scope that encloses the scope of the "this" argument. This constructor requires that the "name" argument reside in a scope that encloses the scope of the "this" argument.

```
@SCJAllowed
@SCJRestricted({javax.safetycritical.annotate.Phase.INITIALIZATION})
public MissionSequencer(PriorityParameters priority,
  StorageParameters storage)
  throws java.lang.IllegalStateException
```

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

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.

Throws IllegalStateException if invoked at an inappropriate time. The only appropriate times for instantiation of a new MissionSequencer are (a) during execution of Safelet.getSequencer by SCJ infrastructure during startup of an SCJ application, and (b) during execution of Mission.initialize by SCJ infrastructure during initialization of a new mission in a Level 2 configuration of the SCJ run-time environment. Note that the static checker for SCJ forbids instantiation of MissionSequencer objects outside of mission initialization, but it does not prevent Mission.initialize in a Level 1 application from attempting to instantiate a MissionSequencer.

Memory behavior: This constructor requires that the "priority" argument reside in a scope that encloses the scope of the "this" argument. This constructor requires that the "storage" argument reside in a scope that encloses the scope of the "this" argument.

Methods

```
@SCJAllowed(javax.safetycritical.annotate.Level.SUPPORT)
protected abstract SpecificMission 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 instantiates and enters the MissionMemory allocation area. The getNextMission method may allocate the returned mission within this newly instantiated MissionMemory allocation area, or it may return a reference to a Mission object that was allocated in some outer-nested MissionMemory 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
public final void requestSequenceTermination( )

Initiate mission termination by invoking the currently running mission's requestTermination method. Upon completion of the currently running mission, this MissionSequencer shall return from its handleAsyncEvent method without invoking getNextMission and without starting any additional missions. Its handleAsyncEvent method will not be invoked again.

Note that requestSequenceTermination does not force the sequence to terminate because the currently running mission must voluntarily relinquish its resources.

Control shall not return from requestSequenceTermination until after the requestTermination method for this mission sequencer's currently running mission has been invoked and control has returned from that invocation. The running mission's requestTermination method is invoked by the requestSequenceTermination method.

It is implementation-defined whether Mission.requestTermination has been called when requestSequenceTermination returns.

@SCJAllowed
public final boolean sequenceTerminationPending( )

returns true if and only if the requestSequenceTermination method has been invoked for this MissionSequencer object.

E.2.18 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 4.7.4 for a discussion of the impact of this on application scheduling.

Constructors

```
@SCJAllowed
@SCJRestricted({javax.safetycritical.annotate.Phase.INITIALIZATION})
public OneShotEventHandler(PriorityParameters priority,
  HighResolutionTime time,
  AperiodicParameters release,
  StorageParameters memory)
```

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 release. 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. A null paramter is equivalent to a relative time of 0.
release — specifies the aperiodic release parameters, in particular the deadline miss handler. A null parameters indicates that there is no deadline associated with this handler.
storage — specifies the storage parameters; it must not be null
Throws IllegalArgumentException IllegalArgumentException if priority, release or memory is null.

Memory behavior: Does not perform memory allocation. Does not allow this to escape local scope. Builds links from this to priority and parameters, so those two arguments must reside in scopes that enclose this.

```
@SCJAllowed(javax.safetycritical.annotate.Level.LEVEL_1)
@SCJRestricted({javax.safetycritical.annotate.Phase.INITIALIZATION})
public OneShotEventHandler(PriorityParameters priority,
  HighResolutionTime time,
  AperiodicParameters release,
```

StorageParameters memory,
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 release. 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. A null paramter is equivalent to a relative time of 0.

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

storage — specifies the storage parameters; it must not be null.

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

Throws IllegalArgumentException IllegalArgumentException if priority, release, or scp is null.

Memory behavior: Does not perform memory allocation. Does not allow this to escape local scope. Builds links from this to priority and parameters, so those two arguments must reside in scopes that enclose this.

Methods

@SCJAllowed(javax.safetycritical.annotate.Level.LEVEL_1)
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)
public javax.realtime.AbsoluteTime getNextReleaseTime(AbsoluteTime dest)

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

dest — The instance of **javax.safetycritical.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 **javax.safetycritical.AbsoluteTime** representing 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 interval parameter (passed at construction time) is based.

Throws IllegalStateException Thrown if this handler has not been started.

@SCJAllowed
@Override
@SCJRestricted({javax.safetycritical.annotate.Phase.INITIALIZATION})
public final void register( )

@SCJAllowed(javax.safetycritical.annotate.Level.LEVEL_1)
public void scheduleNextReleaseTime(HighResolutionTime time)

Change 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 released as if it was created using that type for its time parameter. An absolute time in the passed is equivalent to a relative time 0f (0,0). The rescheduling will take place between the invocation and the return of the method. <P> If scheduleNextReleaseTime is invoked if null </P>

Throws IllegalArgumentException Thrown if time is a negative RelativeTime value or null.

E.2.19 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.

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)
@SCJRestricted({javax.safetycritical.annotate.Phase.INITIALIZATION})
public POSIXRealtimeSignalHandler(PriorityParameters priority,
  AperiodicParameters release,
  StorageParameters storage,
  Happening [] signals)

Constructs an real-time POSIX signalt handler that will be released when the signal is delivered.

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

release — specifies the release parameters for this aperiodic event handler; it must not be null.
storage — specifies the storage requirements for this handler
signals — specifies the range of POSIX real-time signals that releases this handler
Throws IllegalArgumentException IllegalArgumentException if priority, release or event is null.

Memory behavior: Does not perform memory allocation. Does not allow this to escape local scope. Builds links from this to priority and parameters, so those two arguments must reside in scopes that enclose this. Builds a link from this to the event, so the event must reside in memory that encloses this.

E.2.20 CLASS **POSIXSignalHandler**

@SCJAllowed(javax.safetycritical.annotate.Level.LEVEL_1)
public abstract class POSIXSignalHandler

extends javax.safetycritical.ManagedEventHandler

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.

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)
@SCJRestricted({javax.safetycritical.annotate.Phase.INITIALIZATION})
public POSIXSignalHandler(PriorityParameters priority,
  AperiodicParameters release,
  StorageParameters storage,
  Happening [] signals)

Constructs an real-time POSIX signalt handler that will be released when the signal is delivered.

priority — specifies the priority parameters for this periodic event handler. Must not be null.
release — specifies the release parameters for this aperiodic event handler; it must not be null.
storage — specifies the storage requirements for this handler
signals — specifies the range of POSIX real-time signals that releases this handler

Throws IllegalArgumentException IllegalArgumentException if priority, release or event is null.

Memory behavior: Does not perform memory allocation. Does not allow this to escape local scope. Builds links from this to priority and parameters, so those two arguments must reside in scopes that enclose this. Builds a link from this to the event, so the event must reside in memory that encloses this.

E.2.21 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.

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 4.7.4 for a discussion of the impact of this on application scheduling.

Constructors

@SCJAllowed
@SCJRestricted({javax.safetycritical.annotate.Phase.INITIALIZATION})
public PeriodicEventHandler(PriorityParameters priority,
PeriodicParameters release,
StorageParameters storage)

Constructs a periodic event handler.

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, rather it is relative to the point in time when initialization is finished and the timers are started. This argument must not be null.

storage — specifies the storage parameters for the periodic event handler. It must not be null.
Throws IllegalArgumentException IllegalArgumentException if priority, release or memory is null.

Memory behavior: Does not perform memory allocation. Does not allow this to escape local scope. Builds links from this to priority and parameters, so those two arguments must reside in scopes that enclose this.

```
@SCJAllowed(javax.safetycritical.annotate.Level.LEVEL_1)
@SCJRestricted({javax.safetycritical.annotate.Phase.INITIALIZATION})
public PeriodicEventHandler(PriorityParameters priority,
  PeriodicParameters release,
  StorageParameters storage,
  String name)
```

Constructs a periodic event handler.

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. This argument must not be null.
storage — specifies the memory parameters for the periodic event handler. It must not be null.
Throws IllegalArgumentException IllegalArgumentException if priority, release, or scp is null.

Memory behavior: Does not perform memory allocation. Does not allow this to escape local scope. Builds links from this to priority and parameters, so those two arguments must reside in scopes that enclose this.

Methods

```
@SCJAllowed(javax.safetycritical.annotate.Level.LEVEL_1)
public javax.realtime.HighResolutionTime getActualStartTime( )
```

Get the actual start time of this handler. The actual 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. If the actual start time is equal to the effect start time, then the method behaves as if getResquestStartTime() 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 a time parameter based on the clock used to start the timer.

@SCJAllowed(javax.safetycritical.annotate.Level.LEVEL_1)
public javax.realtime.HighResolutionTime getEffectiveStartTime( )

Get the effective start time of this handler. If the clock associated with the start time parameter and the interval 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 interval parameter (passed at construction time) when the handler is actually started.

returns a reference based on the clock associated with the interval parameter.

@SCJAllowed(javax.safetycritical.annotate.Level.LEVEL_1)
public javax.realtime.AbsoluteTime getLastReleaseTime( )

Get the last release time of this handler.

returns a reference to a newly-created **javax.safetycritical.AbsoluteTime** object representing this handlers's last release time, according to the clock associated with the intervale parameter used at construction time.

Throws IllegalStateException Thrown if this timer has not been released since it was last started.

@SCJAllowed(javax.safetycritical.annotate.Level.LEVEL_1)
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 **javax.safetycritical.AbsoluteTime** object. The clock association of the returned time is the clock on which interval parameter (passed at construction time) is based.

Throws ArithmeticException Thrown if the result does not fit in the normalized format.

Throws IllegalStateException Thrown if this handler has not been started.

@SCJAllowed(javax.safetycritical.annotate.Level.LEVEL_1)
public javax.realtime.HighResolutionTime 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 effect the requested start time of this handler if it has not already beend started.

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

@SCJAllowed(javax.safetycritical.annotate.Level.LEVEL_1)
public javax.realtime.AbsoluteTime getnextReleaseTime(AbsoluteTime dest)

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

dest — The instance of **javax.safetycritical.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 **javax.safetycritical.AbsoluteTime** passed as parameter, with time values representing 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 interval parameter (passed at construction time) is based.

Throws ArithmeticException Thrown if the result does not fit in the normalized format.

Throws IllegalStateException Thrown if this handler has not been started.

@SCJAllowed
@Override
@SCJRestricted({javax.safetycritical.annotate.Phase.INITIALIZATION})
public final void register( )

E.2.22 CLASS **PortalExtender**

@SCJAllowed
public abstract class PortalExtender **extends** java.lang.Object

E.2.23 CLASS **PriorityScheduler**

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

The SCJ priority scheduler supports the notion of both software and hardware priorities.

Methods

@SCJAllowed(javax.safetycritical.annotate.Level.LEVEL_1)
@SCJRestricted(maySelfSuspend = false)
public int getMaxHardwarePriority( )

returns the maximum hardware real-time priority supported by this scheduler.

@SCJAllowed(javax.safetycritical.annotate.Level.LEVEL_1)
@SCJRestricted(maySelfSuspend = false)
public int getMinHardwarePriority( )

returns the minimum hardware real-time priority supported by this scheduler.

@SCJAllowed(javax.safetycritical.annotate.Level.LEVEL_1)
public static javax.safetycritical.PriorityScheduler instance( )

returns the SCJ priority scheduler

E.2.24 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.

E.2.25 CLASS **RepeatingMissionSequencer**

@SCJAllowed
public class RepeatingMissionSequencer<SpecificMission **extends** Mission>

extends javax.safetycritical.MissionSequencer

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

The parameter <SpecificMission> allows application code to differentiate between RepeatingMissionSequencers that are designed for use in Level 0 as opposed to missions designed for other compliance levels. For example, a RepeatingMissionSequencer<CyclicExecutive> is known to only run missions that are suitable for execution within a Level 0 run-time environment.

Constructors

```
@SCJAllowed
@SCJRestricted({javax.safetycritical.annotate.Phase.INITIALIZATION},
   maySelfSuspend = false)
public RepeatingMissionSequencer(PriorityParameters priority,
  StorageParameters storage,
  SpecificMission m)
  throws java.lang.IllegalArgumentException,
      java.lang.IllegalStateException
```

Construct a RepeatingMissionSequencer object to oversee execution of the 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.
m — The single mission that runs under the oversight of this RepeatingMissionSequencer.
Throws IllegalStateException if invoked during initialization of a mission whose compliance level is not supported by the implementation.
Throws IllegalArgumentException if any of the arguments equals null.

Memory behavior: This constructor requires that the "priority" argument reside in a scope that encloses the scope of the "this" argument. This constructor requires that the "storage" argument reside in a scope that encloses the scope of the "this" argument. This constructor requires that the "m" argument reside in a scope that encloses the scope of the "this" argument.

```
@SCJAllowed
@SCJRestricted({javax.safetycritical.annotate.Phase.INITIALIZATION},
   maySelfSuspend = false)
public RepeatingMissionSequencer(PriorityParameters priority,
  StorageParameters storage,
  SpecificMission m,
  String name)
  throws java.lang.IllegalArgumentException,
      java.lang.IllegalStateException
```

Construct a RepeatingMissionSequencer object to oversee execution of the 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.

m — The single mission that runs under the oversight of this RepeatingMission-Sequencer.

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

Throws IllegalStateException if invoked during initialization of a mission whose compliance level is not supported by the implementation.

Throws IllegalArgumentException if any of the arguments equals null.

Memory behavior: This constructor requires that the "priority" argument reside in a scope that encloses the scope of the "this" argument. This constructor requires that the "storage" argument reside in a scope that encloses the scope of the "this" argument. This constructor requires that the "m" argument reside in a scope that encloses the scope of the "this" argument. This constructor requires that the "name" argument reside in a scope that encloses the scope of the "this" argument.

```
@SCJAllowed
@SCJRestricted({javax.safetycritical.annotate.Phase.INITIALIZATION},
   maySelfSuspend = false)
public RepeatingMissionSequencer(PriorityParameters priority,
  StorageParameters storage,
  SpecificMission [] missions)
  throws java.lang.IllegalArgumentException,
      java.lang.IllegalStateException
```

Construct a RepeatingMissionSequencer object to oversee execution of the sequence of missions represented by the missions parameter. The Repeating-MissionSequencer runs the sequence of missions identified by its missions array repeatedly, from low to high index position within the array. The constructor allocates a copy of its missions array argument within the current scope.

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

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

missions — An array representing the sequence of missions to be executed under the oversight of this RepeatingMissionSequencer. 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.

Throws IllegalStateException if invoked during initialization of a mission whose compliance level is not supported by the implementation.

Throws IllegalArgumentException if any of the arguments equals null.

Memory behavior: This constructor requires that the "priority" argument reside in a scope that encloses the scope of the "this" argument. This constructor requires that the "storage" argument reside in a scope that encloses the scope of the "this" argument.

```
@SCJAllowed
@SCJRestricted({javax.safetycritical.annotate.Phase.INITIALIZATION},
    maySelfSuspend = false)
public RepeatingMissionSequencer(PriorityParameters priority,
  StorageParameters storage,
  SpecificMission [] missions,
  String name)
  throws java.lang.IllegalArgumentException,
      java.lang.IllegalStateException
```

Construct a RepeatingMissionSequencer object to oversee execution of the sequence of missions represented by the missions parameter. The RepeatingMissionSequencer runs the sequence of missions identified in its missions array repeatedly, from low to high index position within the array. The constructor allocates a copy of its missions array argument within the current scope.

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.

missions — An array representing the sequence of missions to be executed under the oversight of this RepeatingMissionSequencer. 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 RepeatingMissionSequencer will be identified.

Throws IllegalStateException if invoked during initialization of a mission whose compliance level is not supported by the implementation.

Throws IllegalArgumentException if any of the arguments equals null.

Memory behavior: This constructor requires that the "priority" argument reside in a scope that encloses the scope of the "this" argument. This constructor requires that the "storage" argument reside in a scope that encloses the scope of the "this" argument. This constructor requires that the "name" argument reside in a scope that encloses the scope of the "this" argument.

Methods

```
@SCJAllowed(javax.safetycritical.annotate.Level.SUPPORT)
@SCJRestricted({javax.safetycritical.annotate.Phase.INITIALIZATION},
    maySelfSuspend = false)
@Override
protected final SpecificMission 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 Also: javax.safetycritical.MissionSequencer.getNextMission()

E.2.26 CLASS **Services**

@SCJAllowed
public class Services **extends** java.lang.Object

This class provides a collection of static helper methods.

Methods

@SCJAllowed
public static void captureBackTrace(Throwable association)

Captures the stack back trace for the current thread into its thread-local stack back trace buffer and remembers that the current contents of the stack back trace buffer is associated with the object represented by the association argument. The size of the stack back trace buffer is determined by the StorageParameters object that is passed as an argument to the constructor of the corresponding Schedulable. If the stack back trace buffer is not large enough to capture all of the stack back trace information, the information is truncated in an implementation-defined manner.

@SCJAllowed
public static
javax.safetycritical.ManagedSchedulable currentManagedSchedulable( )

@returns
a reference to the currently executed ManagedEventHandler or ManagedThread.

@SCJAllowed(javax.safetycritical.annotate.Level.LEVEL_2)
@SCJRestricted(maySelfSuspend = true)
public static void delay(HighResolutionTime delay)

This is like sleep except that it is not interruptible and it uses nanoseconds instead of milliseconds.

delay — is the number of nanoseconds to suspend. if delay is a RelativeTime type then it represents the number of milliseconds and nanoseconds to suspend. If delay is a time in the past, the method returns immediately.

Throws IllegalArgumentException if the clock associated with delay does not drive events.

@SCJAllowed(javax.safetycritical.annotate.Level.LEVEL_2)
@SCJRestricted(maySelfSuspend = true)
public static void delay(**int** ns_delay)

This is like sleep except that it is not interruptible and it uses nanoseconds instead of milliseconds.

ns_delay — is the number of nanoseconds to suspend

@SCJAllowed(javax.safetycritical.annotate.Level.LEVEL_1)
public static int getDefaultCeiling( )

returns the default ceiling priority. The default ceiling priority is the PriorityScheduler.getMaxPriority. It is assumed that this can be changed using a virtual machine configuration option.

@SCJAllowed
public static javax.safetycritical.annotate.Level getDeploymentLevel( )

returns the deployment compliance level

@SCJAllowed(javax.safetycritical.annotate.Level.LEVEL_0)
@SCJRestricted(maySelfSuspend = false)
public static
javax.realtime.AffinitySet[] getSchedulingAllocationDoamins( )

@SCJAllowed(javax.safetycritical.annotate.Level.LEVEL_0)
@SCJRestricted(maySelfSuspend = false)
public static void nanoSpin(**int** nanos)

Busy wait in nano seconds.

nanos —

@SCJAllowed(javax.safetycritical.annotate.Level.LEVEL_1)
@SCJRestricted({javax.safetycritical.annotate.Phase.INITIALIZATION})
public static void setCeiling(Object O, **int** pri)

Sets the ceiling priority of object O The priority pri can be in the software or hardware priority range. Ceiling priorities are immutable.

Throws IllegalThreadStateException if called outside the mission phase

@SCJAllowed(javax.safetycritical.annotate.Level.LEVEL_0)
@SCJRestricted(maySelfSuspend = false)
public static void spin(HighResolutionTime delay)

Busy wait spinning loop (until now plus delay).

delay — If delay is a RelativeTime type then it represents the number of milliseconds and nanoseconds to suspend. If delay is a time in the past, the method returns immediately.

E.2.27 CLASS StorageParameters

@SCJAllowed
public final class StorageParameters

extends javax.realtime.MemoryParameters

StorageParameters provide storage size parameters for ISRs and schedulable objects in a ManagedSchedulable: event handlers, threads, and sequencers. A StorageParameters object is passed as a parameter to the constructor of mission sequencers and other SCJ schedulable objects.

Constructors

@SCJAllowed
public StorageParameters(**long** totalBackingStore,
long [] sizes,
int messageLength,
int stackTraceLength,
long maxMemoryArea,
long maxImmortal,
long maxMissionMemory)

This is the primary constructor for a StorageParameters object, permitting specification of all settable values.

totalBackingStore — size of the backing store reservation for worst-case scope usage by the associated ManagedSchedulable: object, in bytes.

sizes — is an array of parameters for configuring VM resources such as native stack or Java stack size. The meanings of the entries in the array are vendor specific. The array passed is not stored in the object.

messageLength — memory space in bytes dedicated to the message associated with this ManagedSchedulable object's ThrowBoundaryError exception, plus references to the method names/identifiers in the stack backtrace.

stackTraceLength — is the number of elements in the StackTraceElement array dedicated to stack backtrace associated with this StorageParameters object's ThrowBoundaryError exception.

maxMemoryArea — 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.

maxMissionMemory — is the maximum amount of memory in the mission memory area required by the associated schedulable object.

Throws IllegalArgumentException if any value other than positive. zero, or NO_MAX is passed as the value of maxMemoryArea or maxImmortal.

@SCJAllowed
public StorageParameters(**long** totalBackingStore,
  long [] sizes,
  long maxMemoryArea,
  long maxImmortal,
  long maxMissionMemory)

This is the secondary constructor for a StorageParameters object, permitting specification of backing size and an array of implementation-defined memory sizes.

totalBackingStore — size of the backing store reservation for worst-case scope usage in bytes.

sizes — is an array of parameters for configuring VM resources such as native stack or java stack size. The meaning of the entries in the array are vendor specific. The array passed in is not stored in the object.

maxMemoryArea — 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.

maxMissionMemory — is the maximum amount of memory in the mission memory area required by the associated schedulable object.

Throws IllegalArgumentException if any value other than positive. zero, or NO_MAX is passed as the value of maxMemoryArea or maxImmortal.

E.2.28 CLASS **ThrowBoundaryError**

@SCJAllowed
public class ThrowBoundaryError

extends javax.realtime.ThrowBoundaryError

One ThrowBoundaryError is preallocated for each Schedulable in its outermost private scope.

Constructors

@SCJAllowed @SCJRestricted(maySelfSuspend = false)
public ThrowBoundaryError( )

Allocates an application- and implementation-defined amount of memory in the current scope (to represent the stack backtrace). Shall not copy "this" to any instance or static field.

Memory behavior: This constructor may allocate objects within the currently active MemoryArea.

Methods

@SCJAllowed @SCJRestricted(maySelfSuspend = false, mayAllocate = false)
public java.lang.**Class**> getPropagatedExceptionClass( )

returns a reference to the Class of the exception most recently thrown across a scope boundary by the current thread. Performs no allocation. Shall not copy this to any instance or static field.

@SCJAllowed @SCJRestricted(maySelfSuspend = false)
public java.lang.String getPropagatedMessage( )

returns a newly allocated String object and its backing store to represent the message associated with the thrown exception that most recently crossed a scope boundary within this thread.
The original message is truncated if it is longer than the length of the thread-local \texttt{StringBuilder} object, which length is specified in the \texttt{Storage\-Con\-fig\-ura\-tion\-Pa\-ra\-meters} for this \texttt{Schedulable}.
Shall not copy "this" to any instance or static field.

Memory behavior: This constructor may allocate objects within the currently active MemoryArea.

@SCJAllowed @SCJRestricted(maySelfSuspend = false)
public java.lang.StackTraceElement[] getPropagatedStackTrace( )

returns returns a newly allocated 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 ThrowBoundaryError object. Shall not copy "this" to any instance or static field.
Most commonly, System.captureStackBacktrace() is invoked from within the constructor of java.lang.Throwable. getPropagatedStackTrace() returns a representation of this thread-local back trace information.
Under normal circumstances, this stack back trace information corresponds to the exception represented by this ThrowBoundaryError object. However, certain execution sequences may overwrite the contents of the buffer so that the stack back trace information so that the stack back trace information is not relevant.

Memory behavior: This constructor may allocate objects within the currently active

MemoryArea.

@SCJAllowed @SCJRestricted(maySelfSuspend = false, mayAllocate = false)
public int getPropagatedStackTraceDepth( )

returns the number of valid elements stored within the StackTraceElement array to be returned by getPropagatedStackTrace. Performs no allocation. Shall not copy this to any instance or static field. array to be returned by getPropagatedStackTrace().

Appendix F

Javadoc Description of Package javax.safetycritical.annotate

Package Contents *Page*

F.1 Interfaces

F.2 Classes

F.2.1 CLASS **Level**

@SCJAllowed
public class Level **extends** java.lang.**Enum**

Methods

@SCJAllowed
public static javax.safetycritical.annotate.Level getLevel(String value)

@SCJAllowed
public abstract int value( )

F.2.2 CLASS **Phase**

@SCJAllowed
public class Phase **extends** java.lang.**Enum**

Appendix G

Javadoc Description of Package javax.safetycritical.io

Package Contents *Page*

G.1 Interfaces

G.2 Classes

G.2.1 CLASS ConnectionFactory

@SCJAllowed
public abstract class ConnectionFactory **extends** java.lang.Object
A factory for creating user defined connections.

Constructors

@SCJAllowed
public ConnectionFactory(String name)

Create a connection factory.

name — Connection name used for connection request in Connector.

Methods

@SCJAllowed
public abstract javax.microedition.io.Connection create(String url)
throws java.io.IOException

Create of 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
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
public final java.lang.String getServiceName( )

Return the service name for a connection factory.

returns service name.

@SCJAllowed
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.

G.2.2 CLASS **ConsoleConnection**

@SCJAllowed
public class ConsoleConnection
  implements javax.microedition.io.StreamConnection
  extends java.lang.Object

A connection for the default I/O device.

Constructors

@SCJAllowed
public ConsoleConnection(String name)
  throws javax.microedition.io.ConnectionNotFoundException

Create a new object of this type.

Methods

@Override @SCJAllowed
public void close( )

Closes this console connection.

@Override @SCJAllowed
public java.io.InputStream openInputStream( )

returns the input stream for this console connection.

@Override @SCJAllowed
public java.io.OutputStream openOutputStream( )

returns the output stream for this console connection.

G.2.3 CLASS **SimplePrintStream**

@SCJAllowed
public class SimplePrintStream **extends** java.io.OutputStream

A version of OutputStream that can format a CharSequence into a UTF-8 byte sequence for writing. Issue: kelvin inserted some annotations and added the close method. not sure if expert group agrees with these improvements. recent emails on this topic to the mailing list are going unanswered.

Constructors

@SCJAllowed
@SCJRestricted({javax.safetycritical.annotate.Phase.INITIALIZATION})
public SimplePrintStream(OutputStream **stream**)

stream — to use for output.

Methods

@SCJAllowed
public boolean checkError( )

returns indicates whether or not an error occured.

@SCJAllowed
protected void clearError( )

@SCJAllowed
@SCJRestricted({javax.safetycritical.annotate.Phase.CLEANUP})
public void close( )

@SCJAllowed
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 format:

1 the null byte '\\u0000' is encoded in two bytes rather than in one, so the encoded string never has any embedded nulls; and </li>

2 only the one, two, and three byte encodings are used. </li>

Throws IOException.

@SCJAllowed
public void println( )

@SCJAllowed
public void println(CharSequence sequence)

@SCJAllowed
protected void setError( )

@SCJAllowed
public void write(**int** b)
  throws java.io.IOException

Appendix H

Annotations for Memory Safety

This Appendix started as a section of the SCJ Chapter on Annotations (see Chapter 9). The SCJ Expert Group, late in the drafting process, decided that these annotations targeting memory safety were not ready for standardization and therefore moved them to this Appendix.
As a result, this Appendix cannot be considered normative, which means that implementations need not implement these annotations, and portable SCJ applications shall not depend on their presence.

H.1 Definitions of Memory Safety Annotations

The three SCJ annotations for memory safety, summarized in Table H.1, are as follows.

H.1.1 Scope Tree

An SCJ application contains a finite set of scoped areas; each scoped area has a name and a parent. Scope names must be unique. The scopes and their parent relation must define a well formed *scope tree* rooted at IMMORTAL, the distinguished parent of all scopes.

H.1.2 @DefineScope Annotation

@DefineScope annotation is used to define the *scope tree*. It has two arguments, the symbolic name of the new scope and of its parent scope. The annotation shall be used only on declaration of classes that have an associated scope (for instance, subclasses of the MissionSequencer and Schedulable classes). Annotations required for classes implementing the Runnable interface are:

1. when used for enterPrivateMemory(), the class shall be also annotated with @DefineScope.

Annotation	Where	Arguments	Description
@DefineScope	Any	*Name*	Define a new scope.
@Scope	Class	*Name*	Instances are in named scope.
		CALLER	Can be instantiated anywhere.
	Field	*Name*	Object allocated in named scope.
		UNKNOWN	Allocated in unknown scope.
		THIS	Allocated enclosing class' scope.
	Method	*Name*	Returns object in named scoped.
		UNKNOWN	Returns object in unknown scope.
		CALLER	Returns object in caller's scope.
		THIS	Returns object in receiver's scope.
	Variable	*Name*	Object allocated in named scope.
		UNKNOWN	Object in an unknown scope.
		CALLER	Object in caller's scope.
		THIS	Object in receiver's scope.
@RunsIn	Method	*Name*	Method runs in named scope.
		CALLER	Runs in caller's scope.
		THIS	Runs in receiver's scope.

Table H.1: Annotation summary. Default values in bold.

2. when used for executeInArea(), the class shall be annotated with @DefineScope which refers to an already existing scope and mirrors the @DefineScope annotation used to define this scope.

Furthermore, the @DefineScope annotation shall be added to variable declarations holding ScopedMemory objects. The annotation has the form @DefineScope(name="A", parent="B") where A is the symbolic name of the scope represented by the object and B is the name of the direct ancestor of the scope.

H.1.3 @Scope Annotation

@Scope annotations can be attached to class declarations to constrain the scope in which all instances of that class are allocated. The annotation has the form @Scope("A") where A is the name of a scope introduced by @DefineScope. All methods in the class run in the specified scope by default.
Annotating a field, local or argument declaration constrains the object referenced by that field to be in a particular scope.
Lastly, annotating a method declaration constrains the value returned by that method. Inner classes that are static are independent from the @Scope annotation on the enclosing classes. Non-static inner classes must preserve and restate the @Scope annotation of the enclosing class.

H.1.4 Scope IMMORTAL, CALLER, THIS, and UNKNOWN

The special scope name IMMORTAL is used to denote the singleton instance of ImmortalMemory.

The CALLER, THIS and UNKNOWN scope values can be used in @Scope annotations to increase code reuse. A reference that is annotated CALLER is allocated in the same scope as the allocation context (more on the allocation context in Section H.2). Classes may be annotated CALLER to denote that instances of the class may be allocated in any scope.

References annotated THIS point to objects allocated in the same scope as the receiver (i.e. the value of this) of the current method.

Lastly, UNKNOWN is used to denote unconstrained references for which no static information is available.

H.1.5 @RunsIn Annotation

The @RunsIn annotation can be annotated on a method, In this case, it specifies the context for that particular method, overriding any annotations on its enclosing type. This can be used, for example, to annotate event handlers, which always execute its event handling code in a different scope from which it was allocated. This annotation follows the same form as @Scope.

An argument of CALLER indicates that the method is scope polymorphic and that it can be invoked from any scope. In this case, the arguments, local variables, and return value are by default assumed to be CALLER. If the method arguments or returned value are of a type that has a scope annotation, then this information is used by the Checker to verify the method. If a variable is labeled @Scope(UNKNOWN), the only methods that may be invoked on it are methods that are labeled @RunsIn(CALLER). @RunsIn(THIS) denotes a method which runs in the same scope as the receiver.

H.1.6 Default Annotation Values

For class declarations, the default value is @Scope(CALLER). This is also the annotation on Object. This means that when annotations are omitted classes can be allocated in any scope (and thus are not tied to a particular scope). Local variables and arguments default to CALLER as well. For fields, it is assumed by default that they infer to the same scope as the object that holds them, i.e. their default is THIS. Instance methods have a default @RunsIn(THIS) annotation. The Table H.2 summarizes the values of default annotations for all the source-code elements. Consider the following:

- For @Scope(Name) classes:
  - The unannotated fields and method/constructor parameters of unannotated types are by default @Scope(Name).

Class	Constructor	Constructor Parameters	Field
@Scope(Name)	@RunsIn(Name) @Scope(Name)	@Scope(Name)	@Scope(Name)
@Scope(CALLER)	@RunsIn(CALLER) @Scope(CALLER)	@Scope(THIS)[a]	@Scope(THIS)

Class	Method	Method Parameters	Local Variable
@Scope(Name)	@RunsIn(Name) @Scope(Name)	@Scope(Name)	@Scope(Name)
@Scope(Name)	@RunsIn(CALLER) @Scope(CALLER)	@Scope(CALLER)	@Scope(CALLER)
@Scope(CALLER)	@RunsIn(THIS) @Scope(THIS)	@Scope(THIS)[b]	@Scope(THIS)[c]
@Scope(CALLER)	@RunsIn(CALLER) @Scope(CALLER)	@Scope(CALLER)	@Scope(CALLER)[d]

[a]Where THIS refers to the enclosing class, a parameter from caller's scope is expected to be passed in.

[b]Where THIS refers to the enclosing class, at the caller's side the scope of the parameter must be the same as the scope of the method invocation receiver.

[c]Because the enclosing method is @RunsIn(THIS).

[d]Because the enclosing method is @RunsIn(CALLER).

Table H.2: Summary of default annotations for a class annotated with a named scope and a class annotated as CALLER.

  - Constructors are automatically annotated @RunsIn(Name).
- For @Scope(CALLER) classes:
  - Constructors are automatically annotated @RunsIn(CALLER). This is the only case when the @Scope(CALLER) annotation of the class has an effect on its body, in fact the class' @Scope(CALLER) annotation is considered only during its instantiation.
  - The unannotated fields and method/constructor parameters of unannotated types are by default @Scope(THIS).

Note on the notation: The Table H.2 includes the cases where the class annotation is @Scope(Name). This not only means that the given annotation has a value of a named scope but that this same value must match all the named scope values for the corresponding lines of the table. For example, if the class is annotated @Scope ("S1"), where S1 is a name of a scope, then the default annotations on the class constructors are @Scope ("S1") and @RunsIn ("S1"). The similar notation is adopted in the remainder of this chapter, for every table containing a scope of a value Name, the same scope value must match all the occurrences of the Name on the given line.

H.1.7 Static Fields and Methods

The static constructors are treated as implicitly annotated @RunsIn(IMMORTAL). Static fields are treated as annotated @Scope(IMMORTAL). Thus, static variables follow the same rules as if they were explicitly annotated with IMMORTAL. Every static field must have types that are annotated @Scope(IMMORTAL) or are unannotated.
Static methods are treated as being annotated CALLER.
Strings constants are immutable and therefore are treated as CALLER.

H.1.8 Overriding annotations

The following rules apply for overriding of the memory safety annotations:

1. Class annotation overriding rules:
   (a) @DefineScope annotation cannot be overriden nor restated.
   (b) Subclasses must preserve the @Scope annotation. A subclass of a class annotated with a named scope must retain the exact same scope name. A subclass of a class annotated CALLER may override this with a named scope.
   (c) s, if the class that the method belongs to is annotated @Scope(s)
2. Method annotation overriding rules: Any @RunsIn annotation may be overriden. Further rules apply to upcasting of types that have overriden a @RunsIn annotation, see Section H.5.

H.2 Allocation Context

An allocation context of a method is a scope and its value is the first of:

1. CALLER, if the method is static,
2. s, if the method is annotated @RunsIn(s),
3. CALLER, if the method is annotated @RunsIn(CALLER),
4. s, if the method is annotated @RunsIn(THIS) and if the class that the method belongs to is annotated @Scope(s),
5. s if the method has no annotation and the class that the method belongs to is annotated @Scope(s),

6. THIS if the method is annotated @RunsIn(THIS) and if the class that the method belongs to is annotated @Scope(CALLER) or has no annotation,

7. THIS.

For any given expression, its allocation context is the allocation context of the enclosing method.

H.3 Dynamic Guards

Dynamic guards are equivalent to dynamic type checks. They are used to recover the static scope information lost when a variable is cast to UNKNOWN. A dynamic guard is a conditional statement that tests the value of one of two pre-defined methods, allocatedInSame() or allocatedInParent() or, to test the scopes of a pair of references. If the test succeeds, the check assumes that the relationship between the variables holds. The parameters to a dynamic guard are local variables which must be final to prevent an assignment violating the assumption. The following example illustrates the use of dynamic guards.

```
void method(@Scope(UNKNOWN) final List unk, final List cur) {
  if (ManagedMemory.allocatedInSame(unk, cur)) {
    cur.tail = unk;
  }
}
```

The method takes two arguments, one List allocated in an unknown scope, and the other allocated in the THIS scope. Without the guard the assignment statement would not be valid, because the relation between the objects' scopes can not be validated statically. The guard allows the Checker to assume that the objects are allocated in the same scope and thus the method is deemed valid. Note that the parameters to allocatedInSame and allocatedInParent must be final, so that the variables cannot be modified to violate the assumption.

H.4 Scope Concretization

The value of polymorphic annotations such as THIS and CALLER can be inferred from the allocation context in certain cases. A concretization function translates THIS or CALLER to a named scope where possible. For instance a variable annotated THIS takes the scope of the enclosing class (if the class has a named scope). An object returned from a method annotated CALLER is concretized to the value of the calling method's @RunsIn which, if it is THIS, can be concretized to the enclosing class' scope. and to a class that is enclosing the method corresponding to the given allocation context. Therefore, let:

A scope concretization function *conc(S,C,AC)* is a function of three parameters where :

- S is a scope value,
- AC is the scope of a given allocation context,
- C is the scope of a class enclosing the given allocation context.

and returns one of the following:

- UNKNOWN if S has a value UNKNOWN,
- Name, where Name is some named scope, if either
    - S represents the value Name,
    - S is THIS and C is Name.
- THIS if S is THIS and C is CALLER.
- Name, where Name is some named scope, and S is CALLER and AC is Name.
- CALLER if S is CALLER and AC is CALLER,
- *conc(THIS,C,AC)* if S is CALLER and AC is THIS.

Note that :

- The concretization function does not necessarily yield a named scope.
- While CALLER can be concretized to THIS, the THIS scope can never be concretized to CALLER.
- Concretization of the method's @RunsIn and @Scope annotations is automatically handled by the default annotations rules presented in Section H.1.6. The THIS scope is concretized to Name if the enclosing class has a @Scope annotation, otherwise stays THIS. The CALLER scopes on methods cannot be further concretized.

H.4.1 Equality of two scopes

We say that **two scopes are equal** if they are identical after concretization. The equality can also be expressed by the == operator.

H.5 Scope of an Expression

Every expression must have a scope, if the scope of an expression cannot be determined, the expression is deemed invalid.
The discussion in this section is based on the scope concretization rules presented in Sec.H.4 and thus all the scope values discussed are already concretized to their most concrete value (i.e. scopes THIS and CALLER cannot be further concretized to a named scope).

H.5.1 Simple expressions

To determine the scope of a simple expression, Table H.3 lists all possible cases. For a simple expression, the final scope of an expression is then determined as a concretization function applied to a corresponding valid scope value of the basic expression.

Simple Expression	Result Scope
static expr.	IMMORTAL
enum types	IMMORTAL
string concatenation	*conc(*CALLER*)*
string literal	*conc(*CALLER*)*
this. or super.	*conc(*THIS*)*
local variable	Name/*conc(*THIS/CALLER*)*/UNKNOWN

Table H.3: Scope of a basic expression.

Considering the table, note that:

- **Local Variables:**
  - Local variables, unlike fields and parameters, may have no particular scope associated with them when they are declared and are of a type that is unannotated. Therefore variable is bound to the scope of the right-hand side expression of its first assignment. In the following example

    ```
    Integer myInt = new Integer();
    ```
    if the containing method is @RunsIn(CALLER), myInt is bound to @Scope(CALLER) while the variable itself is still in lexical scope. In other words, it is as if myInt had an explicit @Scope(CALLER) annotation on its declaration.
  - Once a scope is associated with a given variable, it cannot be changed. For example, it would be illegal to have the following assignment in the method body once myInt was already bound to @Scope(CALLER):

    ```
    myInt = Integer.MAX_INT;
    ```
- **String Concatenation:** The concatenation of the two operand strings results in a new string with a scope value of *conc(*CALLER*)*. The scopes of the operand strings do not have any influence on the scope of the resulting string.

H.5.2 Field access

Consider a field access expression e1.f, let:

- S1 be the scope of expression e1,

- S2 be the scope of the field f.

Then, **the scope of an expression** e1.f is S and all its possible values are listed in Tab. H.4.

S1	S2	S
THIS	THIS	THIS
Name1	Name2	Name2
Name	THIS	Name
CALLER	THIS	CALLER
any	UNKNOWN	UNKNOWN
UNKNOWN	Name	Name
UNKNOWN	THIS	UNKNOWN

Table H.4: Scope of a field access expression.

H.5.3 Assignment expressions

Consider assignment expression e1= e2, let :

- S1 be the scope of expression e1, and
- S2 be the scope of expression e2.

Then this assignment expression is valid iff one of the following holds:

1. S1 == S2, or
2. S1 == UNKNOWN, or
3. If the expression e1 is in a form e3.f where
    - e3 is an expression and f is a field, and
    - S3 is the scope of the field access expression e3.f.

    Then the assignment is valid iff:

    (a) S3 == S2, or

    (b) f is UNKNOWN and the expression is protected by the dynamic guard MemoryArea.allocatedInParent(x.f,y), or

    (c) e1.f is THIS and the expression is protected by the dynamic guard MemoryArea.allocatedInSame(x.f,y).

H.5.4 Cast expression

A cast expression (C) e may refine the scope of an expression from an object annotated with CALLER, THIS, or UNKNOWN to a named scope. For example, casting a variable declared @Scope(UNKNOWN)Object to C entails that the scope of expression will be that of C. Casts are restricted so that no scope information is lost. Therefore, consider a cast expression:

```
(A) e;
```

Let:

- the class A be declared as @Scope(S1) class A {...},
- the class B be declared as @Scope(S2) class B extends A {...},
- the type of the expression e be B,
- AC be the scope of the allocation context of the method enclosing the cast expression.

then, the cast expression is valid iff one of the following applies:

- S1 == S2,
- S1 == CALLER and S2 == AC,

A scope of this cast expression is *conc(S1)*.
@RunsIn overriding rule: The following rule related to overriding of the @RunsIn annotation applies for casts:

- Cast is forbidden if the subtype overrides the @RunsIn annotation on a method of the supertype and the method is not annotated SUPPORT.

H.5.5 Method invocation

Consider a method invocation e1.m(...,e2,...), let:

- AC be the scope of the allocation context of the caller,
- ACM be the scope of the allocation context of the invoked method m(),
- T be the scope of the expression e1,
- A be the scope of the expression e2,
- P be the concretized scope of the formal parameter from the method's m() declaration corresponding to the actual argument expression e2,
- SM be the concretized value of the @Scope annotation of the method m(),
- S be the scope of this method invocation expression.

Then, such a method invocation is valid iff I. and II. are valid, and the scope of a method invocation is then determined by III.:
I. Method scope check: one of the following must be valid:

1. The method m is *static*, or
2. The scope ACM is parent to the scope AC and the method m() is annotated @SCJRestricted(mayAllocate=false), or
3. One of the valid cases listed in the Table H.5 applies.

ACM	T	AC
CALLER	any	any
Name	any	Name
THIS	Name	Name
THIS	THIS	THIS
THIS	CALLER	THIS
THIS	CALLER	CALLER

Table H.5: Valid method invocation

Note the following:

(a) The cases where the ACM is CALLER or Name are trivial to resolve.

(b) The only non-trivial case is when the ACM == THIS and it cannot be further concretized because its enclosing class is CALLER. In this case, T == THIS or CALLER and the following applies:

i. if AC == CALLER, then:

A. If T == THIS then this is invalid method call.

B. If T == CALLER then this is valid because AC == T == CALLER == THIS.

ii. If AC == THIS then this method call is valid because T == THIS == CALLER.

II. Method parameter check: assignment of method parameters must be valid, therefore, one of the cases listed in Tab. H.6 must apply.
III. Scope of a method invocation expression: For a valid method invocation expression, all the possible scope values S of such an expression are listed in Tab. H.7.

H.5.6 Allocation expression

Consider an allocation expression new C(y), let:

- A is the scope of an expression y,
- P is a concretized scope of a formal parameter from the constructor declaration corresponding to the actual argument expression y,

P	A	AC	T
Name	Name	any	any
THIS	Name	any	Name
THIS	THIS	THIS	CALLER
THIS	CALLER	CALLER	CALLER
THIS	Name	Name	CALLER
THIS	THIS	any	THIS
CALLER	CALLER	CALLER	any
CALLER	Name	Name	any
CALLER	THIS	THIS	any
UNKNOWN	any	any	any
THIS	any	any	UNKNOWN[a]

[a]Must be guarded by a dynamic guard.

Table H.6: Valid parameter assignment

SM	T	AC	S
Name	any	any	Name
THIS	Name	Name	Name
THIS	CALLER	CALLER	CALLER
THIS	THIS or CALLER	THIS	THIS
CALLER	any	CALLER	CALLER
CALLER	any	Name	Name
CALLER	any	THIS	THIS
UNKNOWN	any	any	UNKNOWN

Table H.7: Scope of a method invocation expression

- AC is the scope of the allocation context of the method enclosing the allocation expression.
- S is the scope of the class C.

Then this allocation expression is valid iff the following holds:

1. One of the following must hold:
   - AC == S,
   - S == CALLER (the class C can be instantiated anywhere).
2. Constructor parameter assignment must be corresponding to one of the valid cases listed in Tab. H.8.

and, **the scope of an allocation expression** is conc(S).
Further, the following rules apply in general for any field or variable declaration:

P	A	AC
Name	Name	any
THIS or CALLER	Name	Name
THIS or CALLER	CALLER	CALLER
THIS or CALLER	THIS	THIS
UNKNOWN	any	any

Table H.8: Valid parameter assignments in constructors.

- A variable or field declaration, C x, is valid if the allocation context is the same or a child of the @Scope of C. Consequently, classes with no explicit @Scope annotation cannot reference classes which are bound to named scopes, because THIS may represent a parent scope.
- By default, the allocation context of an array T[] is the same as that of its element class, T.
- Primitive arrays are considered to be labeled THIS. The default can be overriden by adding a @Scope annotation to an array variable declaration.

H.6 Additional rules and restrictions

The SCJ memory safety annotation system further dictates a following set of rules specific to SCJ API methods.

H.6.1 MissionSequencer and Mission

The MissionSequencer must be annotated with @DefineScope, its getNextMission method has a @RunsIn annotation corresponding to this newly defined scope. Every Mission associated with a particular MissionSequencer is instantiated in this scope and it must have a @Scope annotation corresponding to that scope. Further, MissionSequencer must have @Scope annotation corresponding to the parent scope defined by the @DefineScope annotation.

H.6.2 Schedulables

Each Schedulable must be annotated with a @DefineScope and @Scope annotation. There can be only one instance of a Schedulable class per Mission.

H.6.3 MemoryArea Object Annotation

The annotation system requires every object representing a memory area to be annotated with @DefineScope and @Scope annotations. The annotations allow the checker to statically determine the scope name of the memory area represented by

```
@Scope("M") @DefineScope(name="H", parent="M")
class Handler extends PeriodicEventHandler {

  @RunsIn("H") @SCJAllowed(SUPPORT) void handleAsyncEvent() {
    @Scope(IMMORTAL) @DefineScope(name="M", parent=IMMORTAL)
    ManagedMemory m = (ManagedMemory) MemoryArea.getMemoryArea(this);
    ...
    @DefineScope(name=IMMORTAL,parent=IMMORTAL) @Scope(IMMORTAL)
    ImmortalMemory imm = (ImmortalMemory) ImmortalMemory.instance;
  }
}
```

Figure H.1: Annotating ManagedMemory object example.

the object. This information is needed whenever the object is used to invoke MemoryArea and ManagedMemory API methods, such as newInstance() or executeInArea() and enterPrivateMemory().
The example in Fig. H.1 demonstrates a correct annotation of a ManagedMemory object m. The example shows a periodic event handler instantiated in memory M that runs in memory H. Inside the handleAsyncEvent method a ManagedMemory object representing the scope M is retrieved. As is evident, the variable declaration is annotated with @Scope annotation, expressing in which scope the memory area object is allocated – in this case it is the IMMORTAL memory. Further, the @DefineScope annotation is used to declare which scope is represented by this instance.

H.6.4 EnterPrivateMemory and ExecuteInArea methods

Calls to a scope's executeInArea() method can only be made if the scoped area is a parent of the allocation context. In addition, the Runnable object passed to the method must have a @RunsIn annotation that matches the name of the scoped area. This is a purposeful limitation of what SCJ allows, because the system does not know what the scope stack is at any given point in the program.
Calls to a scope's enterPrivateMemory(size, runnable) method are only valid if the runnable variable definition is annotated with @DefineScope(name="x",parent="y") where x is the memory area being entered and y is a the allocation context. The @RunsIn annotation of the runnable's run() method must be the name of the scope being defined by @DefineScope. The enterPrivateMemory() method cannot be invoked if the allocation context is CALLER.

H.6.5 newInstance

Calls to a scope's newInstance or newArray methods are only valid if the class or element type of the array are annotated to be allocated in target scope or not annotated at all. Similarly, calls to newArrayInArea are only legal if the element type is annotated to be in the same scope as the first parameter or not annotated at all. The expression

ImmortalMemory.instance().newArray(**byte.class**, 10)
should therefore have the scope IMMORTAL. An invocation MemoryArea.newArrayInArea(o, byte.class, 10) is equivalent to calling MemoryArea.getMemoryArea(o).newArray(byte.class, 10). In this case, the scope of the expression is derived from the scope of o.

H.6.6 getCurrent*() methods.

The getCurrent* methods are static methods provided by SCJ API that allow applications to access objects specific to the SCJ infrastructure. The getCurrent*() methods are:

- ManagedMemory.getCurrentManagedMemory(),
- RealtimeThread.getCurrentMemoryArea(),
- MemoryArea.getMemoryArea(),
- Mission.getCurrentMission(),
- MissionManager.getCurrentMissionManager(), and
- Scheduler.getCurrentScheduler().

Typically, an object returned by such a call is allocated in some outer scope; however, there is no annotation present on the type of the object. To explicitly express that the allocation scope of returned object is unknown, the getCurrent*() methods are annotated with @RunsIn(CALLER) and the returned type of such a method call is @Scope(UNKNOWN).

H.7 Validation

The first step to validation of these annotations requires the construction of a reachable class set (RCS); this is the set of all classes that may be manipulated by a SCJ schedulable object. The RCS is constructed by starting with all classes that are annotated @Scope and adding all classes that may be instantiated from run() methods and methods called from run() methods. Therefore, to ensure a successful verification of memory safety annotations, all the source files should be accessible and passed into the Checker at the same time.
We say that an SCJ application is valid if it contains only valid expressions according to the rules described in Sec. H.5 and Sec. H.6.

H.7.1 Disabling Verification of Scope Safety Annotations

The verification of scope safety annotations can be disabled by a compilation parameter -AnoScopeChecks passed to the checker. In this case, only the level compliance annotations and behavior restricting annotations are verified.

H.8 Rationale

The scoped memory area classes extend Java to provide an API for circumventing the need for garbage collection. In Java, the type system guarantees that every access to an object is valid; the garbage collector only recycles objects that are not reachable. Because scoped memory is not garbage collected, it would be possible for the application to retain a reference to a scoped-allocated object, and access the memory after the scope was reclaimed. This could lead to memory corruption and crash the entire virtual machine. In order to ensure memory safety, the RTSJ mandates a number of runtime checks on operations such as memory reads and writes as well as calls to scoped memory enter() and executeInArea(). Exceptions will be thrown if the program performs an operation that may lead to an unsafe memory access. Practical experience with the RTSJ has shown that memory access rules are difficult to get right because the allocation context is implicit and programmers are not used to reasoning in terms of the relative position of objects in the scope hierarchy. In a safety-critical context, these exceptions must never be thrown as they are likely to lead to application failures. Validated programs are guaranteed to never throw any of the following exceptions:

- IllegalAssignmentError occurs when an assignment may result in a dangling pointer. In other words, it occurs when an attempt is made to store a reference to an object where the reference is below the object's memory area in the scope stack.
- ScopedCycleException is thrown when an invocation of enter() on a scope would result in a violation of the single parent rule, which basically states that a scoped memory may only be entered from the same parent scope while it is active.
- InaccessibleAreaException is thrown when an attempt is made to access a memory area that is not on the scope stack (e.g., calling executeInArea() on it).

H.8.1 Memory Safety Annotations Example

The following application-level code snippet illustrates an application of memory safety annotations. The example shows a application-domain source code fragment that defines a MyMission class, where we explicitly declare a scope in which the mission is running by @DefineScope(name="M",parent=IMMORTAL). Furthermore, mission's handler MyHandler is defined to be allocated in mission's memory by @Scope("M"), while running in its own handler's private memory by @RunsIn("H"), defined by the @DefineScope annotation.

```
@Scope(IMMORTAL) @DefineScope(name="M", parent=IMMORTAL)
@SCJAllowed(members=true) class MyMission extends CyclicExecutive {

    @SCJAllowed(SUPPORT) public void initialize() {
        new MyHandler(...);
    }
}

@Scope("M") @DefineScope(name="H", parent="M")
@SCJAllowed(members=true) class MyHandler extends PeriodicEventHandler {

   @SCJAllowed(SUPPORT) @RunsIn("H") public void handleAsyncEvent() {
      ManagedMemory.getCurrentManagedMemory().
           enterPrivateMemory(3000, new Run());
   }
}

@Scope("H") @DefineScope(name="R", parent="H")
@SCJAllowed(members=true) class Run implements SCJRunnable {

     @SCJAllowed(SUPPORT) @RunsIn("R") public void run() {...}
}
```

The application is also expected to define a new scope area any time code enters a child scope. This is illustrated by the Run class that is allocated in MyHandler private memory while running in its own scope. Note the annotations on the Run class; the @DefineScope is used to define a new scope entered by the runnable. Furthermore, the @RunsIn annotation specifies the allocation context of the run() method. Notice that the memory areas form a scope tree with the immortal scope in root.

H.8.2 A Large-Scale Example

In this section we present a Collision Detector (CDx) example and illustrate the use of the memory safety annotations. The classes are written with a minimum number of annotations, though the figures hide much of the logic which has no annotations at all.

The example consists of a periodic task that takes air traffic radar frames as input and predicts potential collisions. The main computation is executed in a private memory area, as the CDx algorithm is executed periodically; data is recorded in a mission memory area. However, because the CDx algorithm relies on positions in the current and previous frame for each iteration, a dedicated data structure, implemented in the Table class, must be used to keep track of the previous positions of each airplane

```
@DefineScope(name="M", parent=IMMORTAL) @Scope("M")
@SCJAllowed(members=true) class CDMission extends Mission {

  @SCJAllowed(SUPPORT) @RunsIn("M") void initialize() {
    new Handler().register();
    MIRun run = new MIRun();
    @Scope(IMMORTAL) @DefineScope(name="M", parent=IMMORTAL)
    ManagedMemory m = (ManagedMemory) MemoryArea.getMemoryArea(this);
    m.enterPrivateMemory(2000, run);
  }
}

@SCJAllowed(members=true)
@Scope("M") @DefineScope(name="MI", parent="M")
class MIRun implements SCJRunnable {
  @SCJAllowed(SUPPORT) @RunsIn("MI") void run() {...}
}
```

Figure H.2: CDx mission implementation.

```
@DefineScope(name="H", parent="M") @SCJAllowed(members=true)
@Scope("M") class Handler extends PeriodicEventHandler {

  Table st;

  @SCJAllowed(SUPPORT) @RunsIn("H") void handleAsyncEvent() {
    Sign s = ... ;
        @Scope("M") V3d old_pos = st.get(s);
        if (old_pos == null) {
          @Scope("M") Sign n_s = mkSign(s);
          st.put(n_s);
        } else ...
  }

  @RunsIn("H") @Scope("M") Sign mkSign(@Scope("M") Sign s) {
    @Scope(IMMORTAL) @DefineScope(name="M",parent="IMMORTAL")
    ManagedMemory m = (ManagedMemory) MemoryArea.getMemoryArea(s);

    @Scope("M") Sign n_s = ManagedMemory.newInstance(Sign.class);
    n_s.b = (byte[]) MemoryArea.newArrayInArea(s, byte.class, s.length);
    for (int i : s.b.length) n_s.b[i] = s.b[i];
    return n_s
  }
}
```

Figure H.3: CDx Handler implementation.

```
@SCJAllowed(members=true) @Scope("M") class Table {

  final HashMap map;
  V3d vectors [];
  int counter = 0;
  final VRun r = new VRun();

  @RunsIn(CALLER) @Scope(THIS) V3d get(Sign s) {
    return (V3d) map.get(s);
  }

  @RunsIn(CALLER) void put(final @Scope(UNKNOWN) Sign s) {
    if (ManagedMemory.allocatedInSame(r,s)) r.s = s;
    @Scope(IMMORTAL) @DefineScope(name="M",parent=IMMORTAL)
    ManagedMemory m = (ManagedMemory) MemoryArea.getMemoryArea(this);
    m.executeInArea(r);
  }
}

@SCJAllowed(members=true) @Scope("M") class VRun implements SCJRunnable {

  Sign s;

  @SCJAllowed(SUPPORT) @RunsIn("M") void run() {
    if (map.get(s) != null) return;
    V3d v = vectors[counter++];
    map.put(s,v);
  }
}
```

Figure H.4: CDx Table implementation.

so that the periodic task may reference it. Each aircraft is uniquely represented by its Sign and the Table maintains a mapping between a Sign and a V3d object that represents current position of the aircraft. Because the state table is needed during the lifetime of the mission, placing it inside the persistent memory is the ideal solution.
First, a code snippet implementing the Collision Detector mission is presented in Fig. H.2. The CDMission class is allocated in a scope named similarly and implicitly runs in the same scope. A substantial portion of the class' implementation is dedicated to the initialize() method, which creates the mission's handler and then shows how the enterPrivateMemory() method is used to perform some initialization tasks in a sub-scope using the MIRun class. The ManagedMemory variable m is annotated with @DefineScope and @Scope to correctly define which scope is represented by this object. Further, notice the use of @DefineScope to define a new MI scope that will be used as a private memory for the runnable.
The Handler class, presented in Fig. H.3, implements functionality that will be periodically executed throughout the mission in the handleAsyncEvent() method. The class is allocated in the M memory, defined by the @Scope annotation. The allocation context of its execution is the "H" scope, as the @RunsIn annotations upon the Handler's methods suggest.
Consider the handleAsyncEvent() method, which implements a communication with the Table object allocated in the scope M, thus crossing scope boundaries. The Table methods are annotated as @RunsIn(CALLER) and @Scope(THIS) to enable this cross-scope communication. Consequently, the V3d object returned from a @RunsIn(CALLER)get() method is inferred to reside in @Scope("M"). For a newly detected aircraft, the Sign object is allocated in the M memory and inserted into the Table. This is implemented by the mkSign() method that retrieves an object representing the scope M and uses the newInstance() and newArrayInArea() methods to instantiate and initialize a new Sing object.
The implementation of the Table is presented in Fig. H.4. The figure further shows a graphical representation of memory areas in the system together with objects allocated in each of the areas. The immortal memory contains only an object representing an instance of the MissionMemory. The mission memory area contains the two schedulable objects of the application – Mission and Handler, an instance representing PrivateMemory, and objects allocated by the application itself – the Table, a hashmap holding V3d and Sign instances, and runnable objects used to switch allocation context between memory areas. The private memory holds temporary allocated Sign objects.
The Table class, presented in Fig. H.4 on the left side, implements several @RunsIn(CALLER) methods that are called from the Handler. The put() method was modified to meet the restrictions of the annotation system, the argument is UNKNOWN because the method can potentially be called from any subscope. In the method, a dynamic guard is used to guarantee that the Sign object being passed as an argument is allocated in the same scope as the Table. After passing the dynamic guard, the

Sign can be stored into a field of the VectorRunnable object. This runnable is consequently used to change the allocation context by being passed to the executeInArea(). Inside the runnable, the Sign is then stored into the map that is managed by the Table class. After calling executeInArea(), the allocation context is changed to M and the object s can be stored into the map. Finally, a proper HashMap implementation annotated with @RunsIn(CALLER) annotations is necessary to complement the Table implementation.

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] P. Dibble et. al. *The Real-Time Specification for Java, V1.1*, 2010. 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-178B, RTCA, 1992.

[7] RTCA & European Organisation for Civil Aviation Equipment. *ED12B. Software Considerations in Airborne Systems and Equipment Certification*, December 1992.

[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.