Safety Critical Specification for Java

Version 0.76
8 July 2010
First Release

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)
Joyce Tokar (Self - employed by Pyrrhusoft)
Jan Vitek (Self - employed by Purdue U.)
Andy Wellings (Self - employed by U. of York)

Consulting Group

Robert Allen (Boeing)
Greg Bollella (Oracle)
Eric Arseneau (Self)
Chris Cole (Apogee Software, Inc.)
Arthur Cook (Self - employed by Alion Science & Technology)
Allen Goldberg (Self)
David Hardin (Rockwell Collins, Inc.)
Takeoka Shozo (AXE, Inc.)

Contents

1 Introduction **1**

1.1 Definitions, Background, and Scope 2

1.2 Additional Constraints on Java Technology 5

1.3 Key Specification Terms 7

1.4 Specification Context 7

1.5 Overview of the Remainder of the Document 8

2 Programming Model **9**

2.1 The Mission Concept 9

2.2 Compliance Levels 10

2.2.1 Level 0 11

2.2.2 Level 1 13

2.2.3 Level 2 13

2.3 SCJ Annotations 13

2.4 Use of Asynchronous Event Handlers 16

2.5 Development vs. Deployment Compliance 16

2.6 Verification of Safety Properties 17

3 Mission Life Cycle **19**

3.1 Semantics and Requirements 20

3.2 Level Considerations 24

3.2.1 Level 0 24

3.2.2 Level 1 25

3.2.3 Level 2 25

3.3 API . . . 25
3.3.1 Safelet . . . 25
3.3.2 MissionSequencer . . . 26
3.3.3 Mission . . . 28
3.3.4 Cyclet . . . 30
3.3.5 CyclicSchedule . . . 32
3.3.6 CyclicSchedule.Frame . . . 33
3.3.7 Level0Mission . . . 33
3.3.8 Level0MissionSequencer . . . 34
3.3.9 SingleMissionSequencer . . . 35
3.4 Application Initialization Sequence Diagram . . . 36
3.5 A Sample Level 0 Application . . . 36
3.6 A Slightly More Complex Level 0 Application . . . 41
3.7 Level 2 Example . . . 42
3.7.1 MyLevel2App.java . . . 42
3.7.2 MainMissionSequencer.java . . . 43
3.7.3 PrimaryMission.java . . . 43
3.7.4 CleanupMission.java . . . 44
3.7.5 SubMissionSequencer.java . . . 44
3.7.6 StageOneMission.java . . . 44
3.7.7 StageTwoMission.java . . . 45
3.7.8 MyPeriodicEventHandler.java . . . 45
3.7.9 MyCleanupThread.java . . . 45

4 Concurrency and Scheduling Models 47
4.1 Semantics and Requirements . . . 48
4.2 Level Considerations . . . 49
4.2.1 Level 0 . . . 49
4.2.2 Level 1 . . . 49
4.2.3 Level 2 . . . 50
4.3 The Parameter Classes . . . 50
4.3.1 Class java.safetycritical.StorageParameters . . . 51

4.3.2 Class javax.realtime.ReleaseParameters 53
4.3.3 Class java.realtime.PeriodicParameters 54
4.3.4 Class javax.realtime.AperiodicParameters 55
4.3.5 Class javax.realtime.SchedulingParameters 55
4.3.6 Class javax.realtime.PriorityParameters 56
4.4 Asynchronous Events and their Handlers 56
4.4.1 Class javax.realtime.AsyncEvent 57
4.4.2 Class javax.safetycritical.AperiodicEvent 57
4.4.3 Class javax.realtime.Schedulable 58
4.4.4 Class javax.safetycritical.ManagedSchedulable 58
4.4.5 Class javax.realtime.AsyncEventHandler 60
4.4.6 Class javax.realtime.BoundAsyncEventHandler 60
4.4.7 Class javax.safetycritical.ManagedEventHandler 60
4.4.8 Class javax.safetycritical.PeriodicEventHandler 61
4.4.9 Class javax.safetycritical.AperiodicEventHandler 63
4.5 Threads and Real-Time Threads 64
4.5.1 Class java.lang.Thread 64
4.5.2 Class java.lang.Thread.UncaughtExceptionHandler 66
4.5.3 Class javax.realtime.RealtimeThread 66
4.5.4 Class javax.realtime.NoHeapRealtimeThread 67
4.5.5 Class javax.safetycritical.ManagedThread 67
4.6 Scheduling and Related Activities 69
4.6.1 Class java.safetycritical.CyclicExecutive 69
4.6.2 Class javax.safetycritical.CyclicSchedule 70
4.6.3 Class javax.safetycritical.CyclicSchedule.Frame 70
4.6.4 Class javax.realtime.Scheduler 71
4.6.5 Class javax.realtime.PriorityScheduler 71
4.6.6 Class javax.safetycritical.PriorityScheduler 72
4.6.7 Class javax.realtime.AffinitySet 72
4.6.8 Class jaxax.safetycritical.Services 74
4.7 Rationale 76

4.7.1 Scheduling and Synchronization Issues . . . 77
4.7.2 Multiprocessors . . . 78
4.7.3 Feasibility Analysis and Multi-Processors . . . 79
4.7.4 Impact of Clock Granularity . . . 80
4.7.5 Deadline Miss Detection . . . 80
4.8 Compatibility . . . 81

5 Interaction with External Devices 83
5.1 Happenings and Interrupt Handling . . . 83
5.1.1 Semantics and Requirements . . . 83
5.1.2 Level Considerations . . . 86
5.2 The Happening Class Hierarchy . . . 87
5.2.1 Class javax.realtime.Happening . . . 87
5.2.2 javax.realtime.EventHappening . . . 89
5.2.3 javax.realtime.AutonomousHappening . . . 89
5.2.4 javax.safetycritical.ManagedAutonomousHappening . . . 90
5.2.5 javax.realtime.EventExaminer . . . 91
5.2.6 javax.realtime.ControlledHappening . . . 91
5.2.7 javax.safetycritical.ManagedControlledHappening . . . 92
5.2.8 javax.realtime.InterruptHappening . . . 93
5.2.9 javax.safetycritical.ManagedInterruptHappening . . . 93
5.3 Raw Memory Access . . . 94
5.3.1 Semantics and Requirements . . . 94
5.3.2 Level Considerations . . . 95
5.3.3 javax.realtime.RawMemoryName . . . 95
5.3.4 javax.realtime.RawIntegralAccess . . . 96
5.3.5 javax.realtime.RawIntegralAccessFactory . . . 99
5.3.6 javax.realtime.RawMemory . . . 100
5.4 Rationale . . . 101
5.5 Compatibility . . . 102

6 Input and Output Model **103**
6.1 Semantics and Requirements . . . 103
6.2 Level Considerations . . . 103
6.3 APIs . . . 104
6.3.1 Interface javax.microedition.io.Connection . . . 104
6.3.2 Class javax.microedition.io.Connector . . . 105
6.3.3 Class javax.microedition.io.ConnectionNotFoundException 106
6.3.4 Interface javax.microedition.io.InputConnection . . . 106
6.3.5 Interface javax.microedition.io.OutputConnection . . . 107
6.3.6 Interface javax.microedition.io.StreamConnection . . . 107
6.4 Rationale . . . 107
6.5 Compatibility . . . 107

7 Memory Management **109**
7.1 Semantics and Requirements . . . 109
7.1.1 Memory Model . . . 110
7.2 Level Considerations . . . 110
7.2.1 Level 0 . . . 110
7.2.2 Level 1 . . . 111
7.2.3 Level 2 . . . 111
7.3 Memory related APIs . . . 112
7.3.1 Interface javax.realtime.AllocationContext . . . 112
7.3.2 Interface javax.realtime.ScopedAllocationContext . . . 114
7.3.3 Class javax.realtime.MemoryArea . . . 114
7.3.4 Class javax.realtime.ImmortalMemory . . . 116
7.3.5 Class javax.realtime.ScopedMemory . . . 116
7.3.6 Class javax.realtime.LTMemory . . . 117
7.3.7 Class javax.safetycritical.ManagedMemory . . . 117
7.3.8 Class javax.safetycritical.MissionMemory . . . 118
7.3.9 Class javax.safetycritical.PrivateMemory . . . 118
7.3.10 Class javax.realtime.SizeEstimator . . . 119
7.4 Rationale . . . 120

7.4.1 Nesting Scopes . . . 122
7.5 Compatibility . . . 122

8 Clocks, Timers, and Time 123
8.1 Semantics and Requirements . . . 123
8.1.1 Clocks . . . 123
8.1.2 Time . . . 123
8.1.3 RTSJ Constraints . . . 124
8.2 Level Considerations . . . 124
8.3 API . . . 124
8.3.1 Class javax.realtime.Clock . . . 124
8.3.2 Interface javax.realtime.ClockCallBack . . . 127
8.3.3 Class javax.realtime.HighResolutionTime . . . 128
8.3.4 Class javax.realtime.AbsoluteTime . . . 129
8.3.5 Class javax.realtime.RelativeTime . . . 132
8.4 Rationale . . . 135
8.5 Compatibility . . . 135

9 Java Metadata Annotations 137
9.1 Semantics and Requirements . . . 137
9.1.1 Annotations for Enforcing Compliance Levels . . . 138
9.1.2 Annotations for Restricting Behavior . . . 140
9.1.3 Annotations for Memory Safety . . . 142
9.2 Level Considerations . . . 146
9.3 API . . . 147
9.3.1 Class javax.safetycritical.annotate.SCJRestricted . . . 147
9.3.2 Class javax.safetycritical.annotate.SCJAllowed . . . 147
9.3.3 Class javax.safetycritical.annotate.Level . . . 148
9.3.4 Class javax.safetycritical.annotate.Restrict . . . 148
9.4 Rationale and Examples . . . 148
9.4.1 Compliance Level Annotation Example . . . 149
9.4.2 Memory Safety Annotations Example . . . 151
9.4.3 A Large-Scale Example . . . 152

10 Class Libraries for Safety Critical Applications 155
10.1 Comparison of SCJ with JDK 1.6 java.io . . . 156
10.2 Comparison of SCJ with JDK 1.6 java.lang package . . . 156
10.3 Comparison of SCJ API with JDK 1.6 java.lang.annotation . . . 168
10.4 Comparison of SCJ Safety Critical Java API with JDK 1.6 java.util 169

11 JNI 171
11.1 Semantics and Requirements . . . 171
11.2 Level Considerations . . . 171
11.3 API . . . 171
11.3.1 Supported Services . . . 171
11.3.2 Annotations . . . 173
11.4 Rationale . . . 173
11.4.1 Unsupported Services . . . 173
11.5 Example . . . 175
11.6 Compatibility . . . 176
11.6.1 RTSJ Compatibility Issues . . . 176
11.6.2 General Java Compatibility Issues . . . 176

12 Exceptions 177
12.1 Semantics and Requirements . . . 177
12.1.1 New Functionality . . . 178
12.2 Level Considerations . . . 179
12.3 API . . . 179
12.3.1 Class java.lang.Error . . . 179
12.3.2 Class java.lang.Exception . . . 180
12.3.3 Class java.lang.Throwable . . . 181
12.3.4 Class jaxax.safetycritical.ThrowBoundaryError . . . 182
12.4 Rationale . . . 183
12.5 Compatibility . . . 185
12.5.1 RTSJ Compatibility Issues . . . 185
12.5.2 General Java Compatibility Issues . . . 185

A Javadoc Description of Package java.io **187**
A.1 Interfaces . . . 189
A.1.1 INTERFACE **Closeable** . . . 189
A.1.2 INTERFACE **Flushable** . . . 189
A.1.3 INTERFACE **Serializable** . . . 189
A.2 Classes . . . 190
A.2.1 CLASS **FilterOutputStream** . . . 190
A.2.2 CLASS **IOException** . . . 191
A.2.3 CLASS **InputStream** . . . 192
A.2.4 CLASS **OutputStream** . . . 193

B Javadoc Description of Package java.lang **195**
B.1 Interfaces . . . 202
B.1.1 INTERFACE **Appendable** . . . 202
B.1.2 INTERFACE **CharSequence** . . . 202
B.1.3 INTERFACE **Cloneable** . . . 203
B.1.4 INTERFACE **Comparable** . . . 204
B.1.5 INTERFACE **Deprecated** . . . 204
B.1.6 INTERFACE **Override** . . . 204
B.1.7 INTERFACE **Runnable** . . . 205
B.1.8 INTERFACE **SuppressWarnings** . . . 205
B.1.9 INTERFACE **Thread.UncaughtExceptionHandler** . . . 205
B.2 Classes . . . 206
B.2.1 CLASS **ArithmeticException** . . . 206
B.2.2 CLASS **ArrayIndexOutOfBoundsException** . . . 207
B.2.3 CLASS **ArrayStoreException** . . . 208
B.2.4 CLASS **AssertionError** . . . 209
B.2.5 CLASS **BigDecimal** . . . 211
B.2.6 CLASS **BigInteger** . . . 223
B.2.7 CLASS **Boolean** . . . 232
B.2.8 CLASS **Byte** . . . 235
B.2.9 CLASS **Character** . . . 240

B.2.10 CLASS **Class** 247
B.2.11 CLASS **ClassCastException** 250
B.2.12 CLASS **ClassNotFoundException** 251
B.2.13 CLASS **CloneNotSupportedException** 252
B.2.14 CLASS **Double** 253
B.2.15 CLASS **Enum** 259
B.2.16 CLASS **Error** 261
B.2.17 CLASS **Exception** 263
B.2.18 CLASS **ExceptionInInitializerError** 264
B.2.19 CLASS **Float** 265
B.2.20 CLASS **IllegalArgumentException** 271
B.2.21 CLASS **IllegalMonitorStateException** 273
B.2.22 CLASS **IllegalStateException** 274
B.2.23 CLASS **IllegalThreadStateException** 275
B.2.24 CLASS **IncompatibleClassChangeError** 276
B.2.25 CLASS **IndexOutOfBoundsException** 277
B.2.26 CLASS **InstantiationException** 278
B.2.27 CLASS **Integer** 279
B.2.28 CLASS **InternalError** 286
B.2.29 CLASS **InterruptedException** 287
B.2.30 CLASS **InvocationTargetException** 288
B.2.31 CLASS **Long** 289
B.2.32 CLASS **Math** 297
B.2.33 CLASS **NegativeArraySizeException** 307
B.2.34 CLASS **NullPointerException** 308
B.2.35 CLASS **Number** 309
B.2.36 CLASS **NumberFormatException** 311
B.2.37 CLASS **Object** 311
B.2.38 CLASS **OutOfMemoryError** 314
B.2.39 CLASS **RuntimeException** 315
B.2.40 CLASS **Short** 316

B.2.41 CLASS **StackOverflowError** . . . 321
B.2.42 CLASS **StackTraceElement** . . . 322
B.2.43 CLASS **StrictMath** . . . 324
B.2.44 CLASS **String** . . . 334
B.2.45 CLASS **StringBuilder** . . . 346
B.2.46 CLASS **StringIndexOutOfBoundsException** . . . 353
B.2.47 CLASS **System** . . . 354
B.2.48 CLASS **Thread** . . . 356
B.2.49 CLASS **Throwable** . . . 359
B.2.50 CLASS **UnsatisfiedLinkError** . . . 362
B.2.51 CLASS **UnsupportedOperationException** . . . 363
B.2.52 CLASS **VirtualMachineError** . . . 364
B.2.53 CLASS **Void** . . . 365

C Javadoc Description of Package javax.microedition.io 367
C.1 Interfaces . . . 369
C.1.1 INTERFACE **Connection** . . . 369
C.1.2 INTERFACE **InputConnection** . . . 369
C.1.3 INTERFACE **OutputConnection** . . . 370
C.1.4 INTERFACE **StreamConnection** . . . 370
C.2 Classes . . . 370
C.2.1 CLASS **ConnectionNotFoundException** . . . 370
C.2.2 CLASS **Connector** . . . 371

D Javadoc Description of Package javax.realtime 373
D.1 Interfaces . . . 379
D.1.1 INTERFACE **AllocationContext** . . . 379
D.1.2 INTERFACE **ClockCallBack** . . . 381
D.1.3 INTERFACE **EventExaminer** . . . 382
D.1.4 INTERFACE **PhysicalMemoryName** . . . 382
D.1.5 INTERFACE **RawIntegralAccess** . . . 382
D.1.6 INTERFACE **RawIntegralAccessFactory** . . . 384

D.1.7 INTERFACE **RawMemoryName** . . . 385
D.1.8 INTERFACE **RawScalarAccess** . . . 385
D.1.9 INTERFACE **RawScalarAccessFactory** . . . 385
D.1.10 INTERFACE **Schedulable** . . . 385
D.1.11 INTERFACE **ScopedAllocationContext** . . . 386
D.2 Classes . . . 387
D.2.1 CLASS **AbsoluteTime** . . . 387
D.2.2 CLASS **AffinitySet** . . . 392
D.2.3 CLASS **AperiodicParameters** . . . 394
D.2.4 CLASS **AsyncEvent** . . . 394
D.2.5 CLASS **AsyncEventHandler** . . . 395
D.2.6 CLASS **AutonomousHappening** . . . 395
D.2.7 CLASS **BoundAsyncEventHandler** . . . 396
D.2.8 CLASS **Clock** . . . 396
D.2.9 CLASS **ControlledHappening** . . . 400
D.2.10 CLASS **EventHappening** . . . 401
D.2.11 CLASS **Happening** . . . 402
D.2.12 CLASS **HighResolutionTime** . . . 404
D.2.13 CLASS **IllegalAssignmentError** . . . 408
D.2.14 CLASS **ImmortalMemory** . . . 408
D.2.15 CLASS **InaccessibleAreaException** . . . 409
D.2.16 CLASS **InterruptHappening** . . . 410
D.2.17 CLASS **LTMemory** . . . 412
D.2.18 CLASS **MemoryAccessError** . . . 413
D.2.19 CLASS **MemoryArea** . . . 413
D.2.20 CLASS **MemoryInUseException** . . . 416
D.2.21 CLASS **MemoryScopeException** . . . 417
D.2.22 CLASS **NoHeapRealtimeThread** . . . 417
D.2.23 CLASS **PeriodicParameters** . . . 418
D.2.24 CLASS **PhysicalMemoryManager** . . . 419
D.2.25 CLASS **PriorityParameters** . . . 420

D.2.26 CLASS **PriorityScheduler** 420
D.2.27 CLASS **ProcessorAffinityException** 421
D.2.28 CLASS **RawMemoryAccess** 421
D.2.29 CLASS **RealtimeThread** 424
D.2.30 CLASS **RelativeTime** 425
D.2.31 CLASS **ReleaseParameters** 429
D.2.32 CLASS **Scheduler** 429
D.2.33 CLASS **SchedulingParameters** 429
D.2.34 CLASS **ScopedCycleException** 430
D.2.35 CLASS **SizeEstimator** 430
D.2.36 CLASS **ThrowBoundaryError** 432

E Javadoc Description of Package javax.safetycritical 433
E.1 Interfaces 437
E.1.1 INTERFACE **ManagedSchedulable** 437
E.1.2 INTERFACE **Safelet** 437
E.1.3 INTERFACE **Schedulable** 438
E.2 Classes 439
E.2.1 CLASS **AperiodicEvent** 439
E.2.2 CLASS **AperiodicEventHandler** 440
E.2.3 CLASS **Cyclet** 441
E.2.4 CLASS **CyclicExecutive** 443
E.2.5 CLASS **CyclicSchedule** 445
E.2.6 CLASS **CyclicSchedule.Frame** 446
E.2.7 CLASS **InterruptHandler** 447
E.2.8 CLASS **InterruptHappening** 448
E.2.9 CLASS **Level0Mission** 448
E.2.10 CLASS **Level0MissionSequencer** 449
E.2.11 CLASS **ManagedEventHandler** 450
E.2.12 CLASS **ManagedInterruptHappening** 451
E.2.13 CLASS **ManagedMemory** 452
E.2.14 CLASS **ManagedThread** 453

E.2.15 CLASS **Mission** 454
E.2.16 CLASS **MissionSequencer** 457
E.2.17 CLASS **NoHeapRealtimeThread** 459
E.2.18 CLASS **PeriodicEventHandler** 460
E.2.19 CLASS **PortalExtender** 462
E.2.20 CLASS **PriorityScheduler** 462
E.2.21 CLASS **PrivateMemory** 462
E.2.22 CLASS **Services** 463
E.2.23 CLASS **SingleMissionSequencer** 465
E.2.24 CLASS **StorageConfigurationParameters** 466
E.2.25 CLASS **StorageParameters** 467
E.2.26 CLASS **Terminal** 469
E.2.27 CLASS **ThrowBoundaryError** 470

F Javadoc Description of Package javax.safetycritical.annotate 473
F.1 Interfaces 474
F.2 Classes 474
F.2.1 CLASS **Level** 474
F.2.2 CLASS **Restrict** 474

G Javadoc Description of Package javax.safetycritical.io 477
G.1 Interfaces 478
G.2 Classes 478
G.2.1 CLASS **Connector** 478
G.2.2 CLASS **ConsoleConnection** 479

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
0.76	First Release	JCP July 2010

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 certification paradigms, but is designed specifically to enable applications and conforming implementations to be certified to 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 meets 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

The Real-Time Specification for Java (RTSJ) [**?**] was designed to address the general needs of adapting Java for use in real-time 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 are 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 [**?**] or in Europe ED-12B [**?**]) 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 out-of-memory conditions are precluded.

In the context of Java technology, as compared to other Java application paradigms, this requires a more efficient and smaller set of Java virtual machines and libraries. They must be more efficient and smaller both to enhance their certifiability and to

permit meeting tight safety critical application performance requirements when running in the Java run-time environments and libraries. Additionally, the applications must exhibit freedom from 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 associated definitions used throughout this standard:

Storey [**?**] provides several useful definitions:

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

Some additional definitions from Burns and Wellings [**?**] 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.
- *Firm real-time components* have associated time constraints that can be missed occasionally, but where there is no benefit from a late response.

In the aviation industry, the DO-178B standard defines 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 may be constructed from Java technology without the aid of this specification.

Other standards have similarly defined levels and also add a probability of such a failure occurring. For example, in IEC 61508 [**?**], 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 [**?**], 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) [**?**] 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 DO-178B were 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 were used to guide key decisions within the Safety Critical Java framework

since 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 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 produce 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-related software is unlikely to result in a successful mission. 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, 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 used when the use of the native RTSJ facilities would obfuscate or add complexity to a conforming application or implementation, or to increase the safety of an interface. Another reason for defining new classes is 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 detect 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 are not required. Many Java and RTSJ classes and methods are omitted. The procedure for starting an application differs from other Java platforms. 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 BoundaryErrorException 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 has the same behavior except that the BoundaryErrorException behaves as if it had been preallocated in per-thread storage.

6. This specification takes no position on whether a Java Virtual Machine is used to execute safety critical applications or the application is compiled to object code and executed using a run-time environment.

1.3 Key Specification Terms

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.

1.4 Specification Context

This specification defines the requirements for JSR-302 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 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 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 JSR-302 conformance. The TCK source code for JSR-302 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 JSR-302 conformance.

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 as follows.

Chapter 2 presents the programming model and introduces the concept of a mission, and the three compliance levels 0, 1, and 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 how a mission (an application or portion of an application) is initialized, run, and terminated.

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, 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 class libraries for SCJ applications.

Chapter 11 presents Java Native Interface (JNI) usage within SCJ applications.

Chapter 12 presents exceptions and the exception model 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

The Real-Time Specification for Java (RTSJ) imposes few limitations on how a developer structures an application, and supports a wide variety of software models in terms of concurrency, packaging, synchronization, memory, etc. Because safety-critical applications must generally conform to rigorous certification requirements, they generally use much simpler programming models that are amenable to certification.

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.

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 as 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 emptying the mission memory and re-entering its initialization mode. If the application designer does not choose to permit restart, it can either avoid terminating the mission, or permit the application to stop all processing.

Conforming implementations are not required to support dynamic class loading. Classses visible within a mission are unexceptionally referenceable. Class initialization must happen before any part of any mission runs, including its initialization phase or its execution 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.

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.

All application processing for a mission occurs in one or more schedulable objects. When a schedulable object is started, its initial memory area is a scoped memory area that is entered when the schedulable object is *released*, and is exited (i.e., emptied) when the schedulable object completes that release. This scoped memory area is not shared with other schedulable objects.

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.

2.2 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, they only refer to different subsets of Java. The specification enables an application compliant with any of the three SCJ levels to satisfy the most stringent of certification objectives, as long as both the application developer and the SCJ infrastructure provider address the targeted certification requirements.

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.

The definition of each level includes the types of schedulable objects (i.e., PeriodicEventHandler, AperiodicEventHandler, no-heap-real-time thread) 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, 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. Figure 2.1 illustrates the execution of a simple Level 0 application, including its memory allocation.

A Level 0 application's schedulable objects shall consist only of a set of PeriodicEventHandler (PEH) instances. Each PEH has a period, priority, and start time relative to the beginning of a major cycle. A schedule of all PEHs is constructed by either the application designer or by an offline tool provided with the implementation.

All PEHs execute under control of 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 syn-

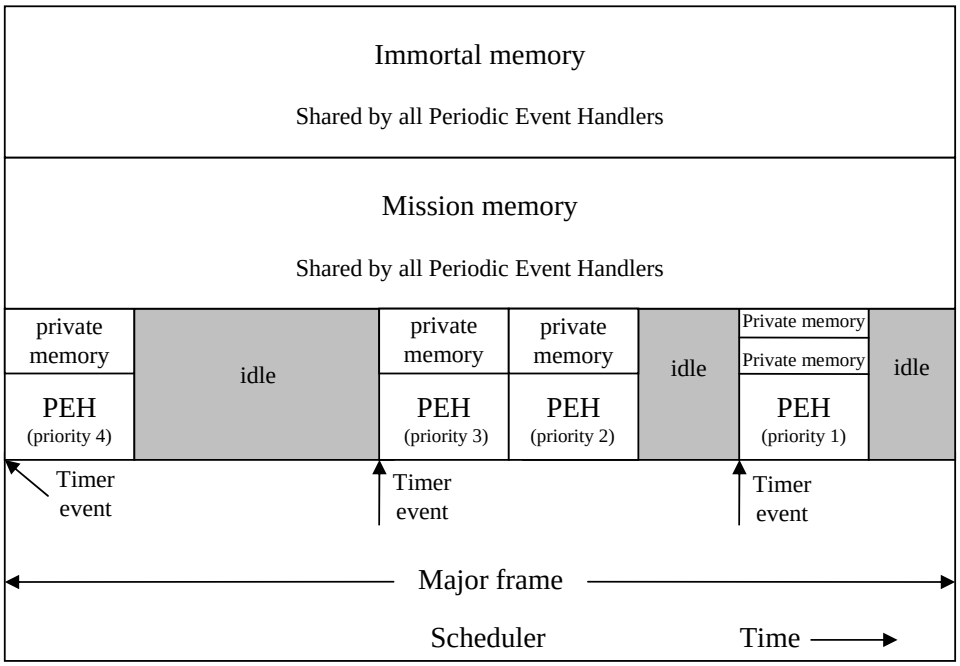

Figure 2.1: Level 0 [Cyclic Executive]

chronization required to safely support its shared objects so the application maintains consistency on a Level 1 or Level 2 implementation as well. The methods Object.wait and Object.notify are not available at Level 0. Applications should also avoid blocking because all of its PEHs are executing in turn in the context of a single thread.

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.

Each PEH has a private scoped memory area, an instance of PrivateMemory, created for it before invocation that will be entered and exited at each invocation. A Level 0 application can create private memory areas directly nested within the provided private memory area, it can enter and exit them, but 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 shall have a private scoped memory area created for it before invocation that will be entered and exited at each invocation. During execution, the PEH or APEH may create, enter, and/or exit one or more other private memory areas, but these memory areas shall not be shared among PEHs or APEHs. Figure 2.2 illustrates the execution of a simple application with a single mission, including its memory allocation.

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 schedulable 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 2 schedulable object shall have a private scoped memory area created for it before 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 schedulable object may create, enter, and/or exit one or more other scoped memory areas, but these scoped memory areas shall not be shared among PEHs or APEHs. Figure 2.3 illustrates the execution of a simple application with one nested mission, including its memory allocation. A Level 2 application may use Object.wait and Object.notify.

2.3 SCJ Annotations

To permit 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 description of these annotations

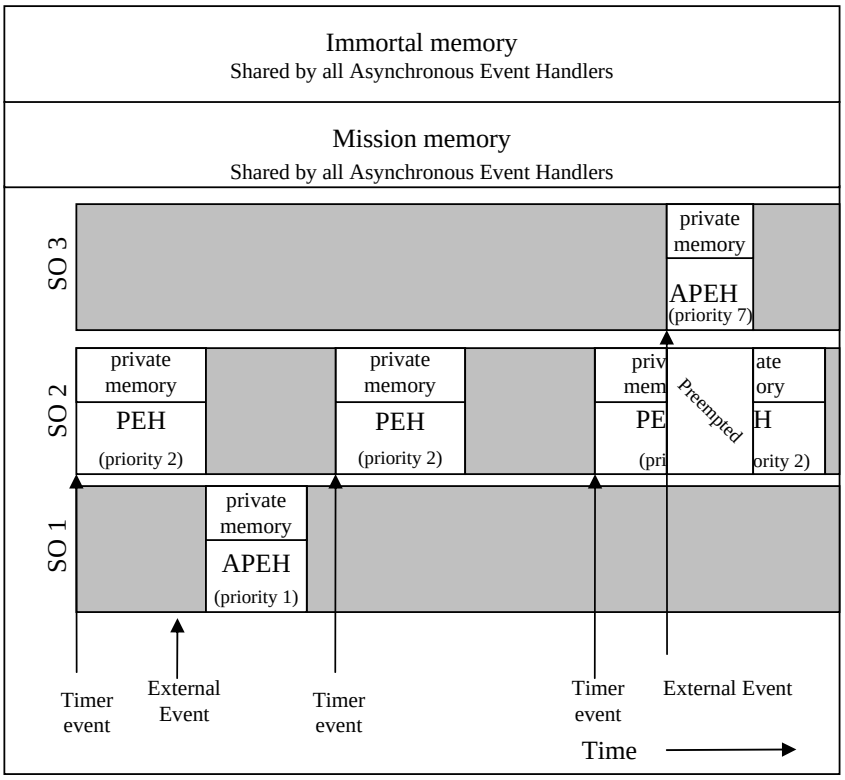

Figure 2.2: Level 1 [Single Mission]

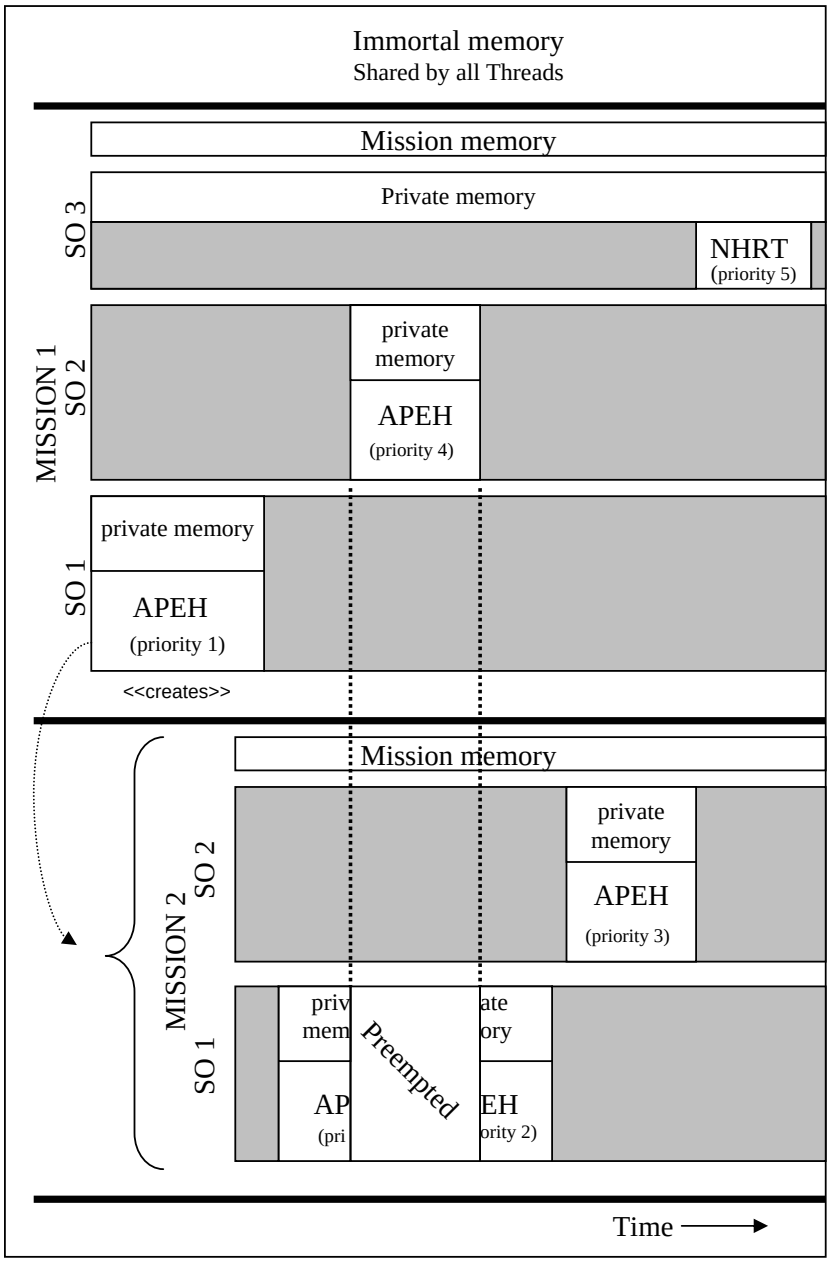

Figure 2.3: Level 2 [Nested Missions]

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.

2.4 Use of Asynchronous Event Handlers

The RTSJ defines two mechanisms for real-time execution: the RealtimeThread and NoHeapRealtimeThread classes, which uses a 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 provides only AsynchronousEventHandlers at Levels 0 and 1, permitting no-heap real-time threads only at Level 2. In particular, this specification permits the following:

- Level 0: periodic AsynchronousEventHandler (class PeriodicEventHandler).
- Level 1: periodic and aperiodic (but not sporadic) AsynchronousEventHandler (classes PeriodicEventHandler and AperiodicEventHandler).
- Level 2: no-heap real-time threads, periodic and aperiodic (but not sporadic) AsynchronousEventHandlers.

The classes PeriodicEventHandler and AperiodicEventHandler are defined by this specification. The application programmer establishes a periodic activity by extending PeriodicEventHandler, overriding the handleAsyncEvent method to perform the per-release processing, and constructing an instance with the desired priority and release parameters. This is different from the style in the RTSJ, which requires associating a periodic AsynchronousEventHandler with a periodic timer.

Sporadic AsynchronousEventHandler are not provided because their management would require the implementation to monitor minimal interarrival time for asynch events. It was determined that this would add excessive complexity with a resulting impact on safety-critical certifiability.

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. On the contrary, for implementations usable only during development, while it is preferable for these requirements to be imposed, a limited number of exceptions are explicitly permitted. These exceptions 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 must be able to tolerate Priority Inheritance behavior without creating functional errors, although its performance characteristics may not be correctly supported.

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 checking for, and handling for such errors in the runtime environment.

Chapter 3

Mission Life Cycle

An SCJ application is a sequence of mission executions. Each Mission is comprised of initialization, execution, and cleanup phases, as illustrated in the following figure.

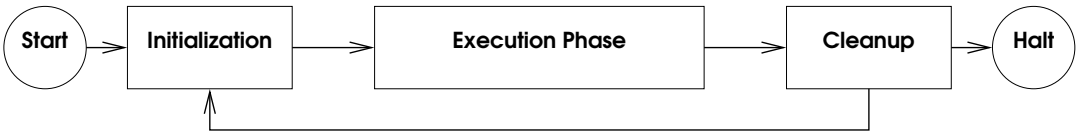

Figure 3.1: Safety Critical Application Phases

Application Initialization: An SCJ application is represented by a user-defined implementation of the Safelet interface. The application defines the setUp method, which may create objects in the ImmortalMemoryArea memory and may perform certain other side effects, such as setting up hardware interrupt handlers and initializing hardware devices.

The application also defines the getSequencer method, which returns the MissionSequencer that represents the application. The MissionSequencer identifies a sequence of user-defined missions, each of which is represented by a class that extends the Mission class.

Mission Initialization: The MissionSequencer arranges to invoke the initialize method for each Mission. This method instantiates and registers all of the ManagedSchedulable objects corresponding to this Mission and it allocates and initializes objects to be shared between these ManagedSchedulable objects. In particular, all ManagedSchedulable objects used by the mission shall be created and registered in initialize.

Mission Execution: The application's missions shall run under the direction of the MissionSequencer in the order that the Missions are returned from the MissionSequencer's implementation of getNextMission. By default, memory allocations take place in the PrivateMemory areas associated with each ManagedSchedulable, but

allocations can also be directed to ImmortalMemoryArea or MissionMemory. Each ManagedSchedulable is free to use additional PrivateMemory areas that can be seen as inner scopes of the MissionMemory. Before a Mission can terminate, it must wait for all of the managedSchedulables associated with the Mission to complete their execution.

Mission Cleanup: The application defines the cleanup method. The cleanup phase can be used to free resources. This is required for missions terminated asynchronously by parent missions. After the cleanup phase is complete, the mission can either be restarted, or replaced by a different Mission, or the entire application can halt, under the direction of the MissionSequencer.

Application Cleanup: Upon termination of the application, the tearDown method is invoked and any resources used by the application, such as external handles to devices, can be freed

Mission Life Cycle Interactions

The runtime environment for a safety critical system differs from an RTSJ system. In RTSJ, the system starts with a normal Java thread in heap memory. In SCJ, the application runs entirely under the control of the SCJ infrastructure, without any involvement of traditional Java threads.

The SCJ run-time environment is started up with a request to execute the SCJ application represented by a particular application-defined implementation of the Safelet interface. This document refers to the user-defined Safelet implementation as *the application*. This chapter describes the life cycle of an application. In particular, it describes the initialization, running, and termination of the application. Differences in the life cycle between Level 0, Level 1 and Level 2 applications are also described.

The application defines itself through the Safelet interface, MissionSequencer and Mission classes. The Safelet implementation defines how the environment starts and terminates the application. The MissionSequencer definition describes transfers of control between the various Mission objects that execute in sequence within the application. Applications running at Level 2 may also create nested missions, which are launched from an outer-nested mission and have their own MissionSequencer for driving a sequence of sub-missions.

3.1 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.

The normal sequence of activities for an application consists of several steps:

1. **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:

   - It is expected that vendors of SCJ implementations will provide a mechanism to specify the priority at which the initialization thread runs relative to other non-Java threads which may be running concurrently on the same hardware.
   - It is expected that vendors of SCJ implementations will provide a mechanism to specify the StorageConfigurationParameters for this initial thread. The resources requested by the StorageConfigurationParameters of the Safelet's MissionSequencer are taken from the StorageConfigurationParameters for this initial thread.
   - The SCJ infrastructure shall load and initialize all classes used by the application before invoking any user-written code associated with Safelet. The mechanism for identifying which classes need to be loaded and specifying a deterministic class initialization order shall be implementation defined.
   - The initial allocation context for the initialization thread is ImmortalMemoryArea.
   - User-defined methods executed by this thread are allowed to introduce PrivateMemory memory regions at their discretion.
   - Once control has transferred to the Safelet's MissionSequencer, the initialization thread shall remain blocked until the MissionSequencer terminates its execution.

   The SCJ infrastructure performs the following sequence of activities:

   - Invokes the setUp method of Safelet to run user-defined initialization code that must precede execution of the MissionSequencer.
   - Invokes the getSequencer method of Safelet and remembers the result, which represents the MissionSequencer object seq that is responsible for running the sequence of missions that comprise this application. The returned object shall reside in ImmortalMemoryArea. The StorageConfigurationParameters resources for MissionSequencer seq are taken from the StorageConfigurationParameters resources for the initialization thread.

- Causes MissionSequencer seq to begin executing. The details of this implementation are not specified in the SCJ specification.
- Blocks the initialization thread until MissionSequencer seq terminates its execution.
- Invokes the tearDown method of Safelet to run user-defined finalization code that must follow termination of the MissionSequencer.

2. **MissionSequencer Behavior:**

   Between execution of getSequencer and tearDown, the SCJ infrastructure arranges for the MissionSequencer to execute the sequence of missions with which it is associated. The typical sequence of the infrastructure activities involving the MissionSequencer consists of the following interactions:

   - In the case that a MissionSequencer is the outermost MissionSequencer for a given SCJ application, the Safelet's execution thread causes the independent thread that is associated with the MissionSequencer to begin executing. The thread's storage resource requirements are specified by the StorageConfigurationParameters argument to the MissionSequencer constructor. These resources are taken from the storage resources of the application's initialization thread. Note that the Safelet's MissionSequencer, which is an extension of BoundAsynchronousEventHandler, is instantiated and starts up outside of an enclosing Mission. This is the only circumstance under which a BoundAsynchronousEventHandler may be instantiated outside of a Mission's initialize method.
   - Alternatively, in the cast that a MissionSequencer nests within a Level 2 Mission, the MissionSequencer must be instantiated and registered within that Mission's initialize code. The thread's storage resource requirements are specified by the StorageConfigurationParameters argument to the MissionSequencer constructor. These resources are taken from the storage resources of the enclosing Mission's MissionSequencer thread. When Mission initialization completes, the SCJ infrastructure arranges to start up the execution of all threads associated with ManagedSchedulable objects, including any MissionSequencer objects that were instantiated and registered during Mission initialization.
   - When the MissionSequencer's bound event handling thread begins to execute, it instantiates a MissionMemory object to hold the first Mission to be executed by the MissionSequencer. The backing store associated with this MissionMemory object is initially sized to represent all of the remaining backing store memory available within the current thread's storage resources.
   - Next, the MissionSequencer's bound event handling thread enters into the newly created MissionMemory area and invokes its own getNext-

Mission 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, should return a reference to a Mission object residing in the MissionMemory area.

- Having selected the Mission that is going to execute first, the MissionSequencer thread now invokes the missionMemorySize method on that Mission. It passes the value returned from missionMemorySize to an invocation of resize on the MissionMemory area. This truncates the size of the MissionMemory's backing store, returning the excess memory to the running thread's backing store so that it can be used to represent PrivateMemory areas.
- The MissionSequencer then arranges to execute the selected Mission and waits for that Mission to complete. The detailed steps that comprise Mission execution are described in the next subsection.
- When the mission finishes its execution, control exits the MissionMemory area allocated above, releasing all of the objects that had been allocated within that MissionMemory, including the Mission object itself.
- The MissionSequencer now instantiates another MissionMemory object to hold the next Mission to be executed by the MissionSequencer. As with the initial MissionMemory object allocated by this MissionSequencer, the backing store associated with this MissionMemory object is initially sized to represent all of the remaining backing store memory available within the backing store associated with the currently running thread.
- The MissionSequencer's event handling thread invokes MissionSequencer.getNextMission to obtain a reference to the next mission to be executed by this MissionSequencer. If getNextMission returns null, the MissionSequencer considers its role to be complete and it terminates by exiting the dedicated MissionMemory area and returning control to its controlling context (the outer-nested Safelet or Mission). Otherwise, the MissionSequencer arranges to execute the selected Mission and control follows the same sequence described above.
- Under certain circumstances, user code may request that a particular MissionSequencer terminate its execution by invoking the MissionSequencer's requestSequenceTermination method. This in turn invokes the currently running mission's requestTermination method, and then causes the MissionSequencer's event handling thread to wait for that Mission to end its execution. Once the Mission ends, the MissionSequencer's event handling thread terminates without another invocation of getNextMission.

3. **Mission Execution:**

   For each Mission executed by the MissionSequencer, the MissionSequencer's event handling thread performs the following actions:

- With MissionMemory as the current allocation area, the MissionSequencer invokes the Mission's initialize method, which is written by the application developer. Note that this method may allocate mission-relevant objects within the MissionMemory. The initialize method may also introduce PrivateMemory scopes for temporary computations. Within the initialize method, application code allocates and registers one or more ManagedEventHandler or ManagedThread or MissionSequencer objects as part of the Mission. Each of these represents an independent thread of control with a backing store memory reservation specified by a StorageConfigurationParameters object. The backing store for each independent thread is obtained by setting aside contiguous segments of the backing store memory that had been previously associated with the MissionSequencer's event handling thread.
- Upon return from initialize, the MissionSequencer's event handling thread starts up the threads associated with each of the ManagedSchedulable objects that had been registered by initialize.
- The MissionSequencer'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 Mission's mission phase terminates, except for small and predictable implementation-defined increments of work associated with arranging to shut down a running Mission.
- When the MissionSequencer's event handling thread detects that the Mission's mission phase has terminated, it invokes the Mission's cleanup method to allow the Mission to finalize mission objects and perform whatever other activities are necessary in order to finish shutting down the Mission.
- Upon return from the cleanup method, the MissionSequencer's event handling thread exits the MissionMemory scope so the memory for the just-completed Mission can itself be reclaimed.

3.2 Level Considerations

3.2.1 Level 0

A Level 0 application shall extend Cyclet, a library class which implements Safelet. The getSequencer method of Cyclet is declared to return a Level0MissionSequencer object. The getNextMission method of Level0MissionSequencer is declared to return a Level0Mission. Thus, the type system enforces that a Level 0 application is comprised only of Level0Mission missions. This is important because the SCJ infrastructure requires that a CyclicSchedule be associated with each Mission in the

Level 0 application.

The Level0Mission subclass must implement the getSchedule method. This method returns a reference to a CyclicSchedule object which represents the static cyclic schedule for the PEHs associated with this Level0Mission. This schedule would typically be generated by vendor-specific tooling, but it can also be generated by hand. The Level 0 example described later in this chapter shows a generated schedule.

3.2.2 Level 1

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

3.2.3 Level 2

A Level 2 application shall implement Safelet, similar to a Level 1 application. An enhanced capability of Level 2 applications is the option to instantiate ManagedThreads and inner-nested MissionSequencers during execution of a Mission's initialize method.

3.3 API

3.3.1 Safelet

Declaration

@SCJAllowed
public interface Safelet

Description

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

Given the implementation app of Safelet that represents a particular SCJ application, the SCJ infrastructure invokes in sequence app.setUp() followed by app.getSequencer().

For the MissionSequencer returned from app.getSequencer(), the SCJ infrastructure arranges for an independent thread to begin executing the code for that MissionSequencer and then waits for that thread to terminate its execution. Upon termination of the MissionSequencer's thread, the SCJ infrastructure invokes app.tearDown().

Methods

```
@SCJAllowed
public MissionSequencer getSequencer()
```

Returns the MissionSequencer responsible for selecting the sequence of Missions that represent this SCJ application. The infrastructure invokes getSequencer to obtain the MissionSequencer that oversees execution of Missions for this application. The returned MissionSequencer resides in ImmortalMemoryArea. Note that MissionSequencer is an extension of BoundAsynchronousEventHandler. The StorageConfigurationParameters resources for the MissionSequencer's bound thread are taken from the StorageConfigurationParameters resources for the Safelet's initialization thread. The initialization infrastructure arranges to start up the corresponding BoundAsynchronousEventHandler and causes its event handling code to execute in the corresponding bound Thread. The event handling code, provided in the MissionSequencer's final handleAsyncEvent() method, begins executing with ImmortalMemoryArea as its current allocation area.

```
@SCJAllowed
public void setUp();
```

The infrastructure invokes setUp before invoking getSequencer. Application developers place code to be executed before the MissionSequencer begins to execute within this method. Upon entry into this method, the current allocation context is ImmortalMemoryArea. User code may introduce nested PrivateMemory areas for temporary computations.

```
@SCJAllowed
public void tearDown();
```

The infrastructure invokes tearDown after the MissionSequencer returned from getSequencer completes its execution. Application developers place code to be executed following MissionSequencer execution within this method. Upon entry into this method, the current allocation context is ImmortalMemoryArea. User code may introduce nested PrivateMemory areas for temporary computations.

3.3.2 MissionSequencer

Declaration

```
@SCJAllowed
```

public abstract class MissionSequencer **extends** BoundAsyncEventHandler

Description

A MissionSequencer runs a sequence of independent or repeated Mission executions.

Constructor

```
@SCJAllowed
@SCJRestricted({INITIALIZATION})
public MissionSequencer(PriorityParameters priority,
                        StorageConfigurationParameters storage)
```

Construct a MissionSequencer to run at the priority and with the memory resources specified by its parameters.

Throws IllegalStateException if not invoked during initialization of a Level 2 Mission or during the infrastructure's invocation of an SCJ application's Safelet.getSequencer method.

Methods

```
@SCJAllowed
protected abstract Mission getNextMission()
```

This method is called by the 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 by the infrastructure, the infrastructure instantiates and enters a MissionMemory, initially sized to represent all available backing store for the currently running thread. Though this method is implemented by the application developer, the expectation is that this method returns a Mission object newly allocated within the default MissionMemory 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 handleAsyncEvent()
```

This method is declared final because the implementation is provided by the vendor of the SCJ implementation and shall not be overridden. This method performs all of the activities that correspond to sequencing of Missions by this MissionSequencer.

```
@SCJAllowed(LEVEL_2)
public final void requestSequenceTermination()
```

Try to finish the work of this mission sequencer soon 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.

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

@SCJAllowed(LEVEL_2)
public final boolean sequenceTerminationPending()

Check if the current Mission is trying to terminate.

Returns true if and only if this MissionSequencer's requestSequenceTermination method has been invoked.

3.3.3 Mission

Declaration

@SCJAllowed
public abstract class Mission

Description

An SCJ application is comprised of one or more Missions. Each Mission is implemented as a subclass of this abstract Mission class. Only Missions that implement Safelet can be started directly. These missions are called primary missions. A primary mission can implement Safelet directly if it is a Level 1 or Level 2 application.

Constructor

@SCJAllowed
public Mission()

Constructor for a Mission. Normally, the infrastructure instantiates a new Mission in the MissionMemory area that is dedicated to that Mission. Upon entry into the constructor, this same MissionMemory area is the current allocation area.

Methods

@SCJAllowed(LEVEL_1)
protected void cleanup()

Method to clean up after an application terminates. The infrastructure calls cleanup after all managedSchedulables associated with this Mission have terminated but before control leaves the dedicated MissionMemory area. The default implementation of cleanup does nothing. User-defined subclasses may override its implementation.

@SCJAllowed()
protected abstract void initialize()

Perform initialization of this Mission. The infrastructure calls initialize after the Mission has been instantiated and the MissionMemory has been resized to match the size returned from this Mission's missionMemorySize method. 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. User-defined subclasses may override its implementation.

The typical implementation of initialize instantiates and registers all managedSchedulables that constitute this Mission. The infrastructure enforces that managedSchedulables can only be instantiated and registered if the infrastructure is currently executing a Mission.initialize method. (A special exception to this rule allows a MissionSequencer to be instantiated if the infrastructure is currently executing Safelet.getSequencer.) The infrastructure arranges to begin executing the managedSchedulables registered by the initialize method upon return from the initialize method.

Besides initiating the associated managedSchedulables, this method may also instantiate and/or initialize certain mission-relevant data structures. Note that objects shared between managedSchedulables typically reside within the MissionMemory scope. Individual managedSchedulables gain access to these objects by consulting reference arguments passed to their constructors from the Mission.initialize method, or by obtaining a reference to the current Mission (by invoking Mission.getCurrentMission method) and coercing this reference to the known Mission subclass. It is better style to pass references to shared objects as constructor arguments, as this allows the same managedSchedulables to be reused as elements of multiple distinct Mission subclasses.

```
@SCJAllowed()
public void requestTermination()
```

This method provides a standard interface for requesting termination of a mission. The description of semantics that follows makes reference to several internal details that are not part of the public SCJ API. This description makes reference to terms that are defined in the Real-Time Specification for Java. An SCJ implementation that does not incorporate the full implementation of the Real-Time Specification for Java shall emulate the relevant capabilities in order to implement the specified semantics. The default implementation has the effect of setting internal state so that subsequent invocations of terminationPending shall return true. Additionally, this method has the effect of (1) disabling all periodic event handlers associated with this Mission, (2) invoking the removeHandler service on each AsyncEvent associated with this Mission for every BoundAsynchronousEventHandler affiliated with the AsyncEvent, and (3) clearing the pending fire count for each of this Mission's event handlers so that the event handler threads can terminate following completion of any event handling code that is currently active.

An application-specific subclass of Mission may override this method in order to insert application-specific code to communicate the intent to shutdown to specific

managedSchedulables. It is especially useful to override requestTermination within Missions that include ManagedThread or nested MissionSequencers as communicating the shutdown request may rely on application-specific protocols. When overriding this method, it is generally advisable to invoke super.requestTermination either before or after executing the application-specific implementation of requestTermination.

@SCJAllowed()
public final void requestSequenceTermination()

Ask for termination of the current Mission and its MissionSequencer. The effect of this method is to invoke requestSequenceTermination on the MissionSequencer that is responsible for execution of this Mission.

@SCJAllowed()
public void terminationPending()

Check if the current mission is trying to terminate.

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

@SCJAllowed()
public final boolean sequenceTerminationPending()

Check if the current MissionSequencer is trying to terminate.

Returns true if and only if the requestSequenceTermination method for the MissionSequencer that controls execution of this Mission has been invoked.

@SCJAllowed()
abstract public long missionMemorySize()

Returns the desired size of the MissionMemory associated with this Mission. Note that the MissionMemory is allocated initially with a very large size, and then is truncated to the size returned from this method, which is invoked by the infrastructure immediately following return from instantiation and construction of this Mission object.

@SCJAllowed
public static Mission getCurrentMission()

Returns the instance of the Mission to which the currently running managedSchedulable belongs.

3.3.4 Cyclet

Declaration

```
@SCJAllowed
public class Cyclet implements Safelet
```

Description

Every Level 0 SCJ application is represented by a Cyclet or a subclass of Cyclet which identifies the outer-most MissionSequencer. This outer-most MissionSequencer takes responsibility for running the sequence of Missions that comprise this SCJ application.

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

Given class app of type Cyclet or a subclass of Cyclet that represents a particular SCJ application, the SCJ infrastructure invokes in sequence app.setUp() followed by app.getSequencer(). For the MissionSequencer returned from app.getSequencer(), the SCJ infrastructure arranges for an independent thread to begin executing the code for that MissionSequencer and then waits for that thread to terminate its execution. Upon termination of the MissionSequencer's thread, the SCJ infrastructure invokes app.tearDown().

Constructor

```
@SCJAllowed(LEVEL_0)
public Cyclet()
```

Construct a Cyclet.

Methods

```
@SCJAllowed
@SCJRestricted({INITIALIZATION})
protected Level0Sequencer getSequencer()
```

Returns the MissionSequencer that oversees execution of Missions for this application.

The default implementation of getSequencer returns a SingleMissionSequencer which runs the Level0Mission represented by getPrimordialMission exactly once. The values for this thread's priority and StorageConfigurationParameters are specified at configuration time in an implementation-defined way.

```
@SCJAllowed
public void setUp()
```

The infrastructure invokes setUp before invoking getSequencer. Application developers place code to be executed before the MissionSequencer begins to execute within this method. Upon entry into this method, the current allocation context is ImmortalMemoryArea. User code may introduce nested PrivateMemory areas for temporary computations.

```
@SCJAllowed
public void tearDown()
```

The infrastructure invokes tearDown after the MissionSequencer returned from getSequencer completes its execution. Application developers place code to be executed following MissionSequencer execution within this method. Upon entry into this method, the current allocation context is ImmortalMemoryArea. User code may introduce nested PrivateMemory areas for temporary computations.

```
@SCJAllowed()
public static Level0Mission getPrimordialMission()
```

At configuration time, the developer has the option of specifying a primordial mission, which is identified by a fully qualified class name. Configuration parameters might be specified as options to the SCJ compiler or linker, or might be specified on the command line that starts up execution of the SCJ application. The SingleMissionSequencer's getNextMission method, which is invoked by the infrastructure with the MissionMemory dedicated to this Mission as the active allocation area, invokes getPrimordialMission. If a primordial mission was specified at configuration time, this method returns a reference to an instance of the primordial mission, allocated in the dedicated MissionMemory area. If no primordial mission was specified, this method returns null.

In the case that this method returns a non-null result, the returned object is allocated at the time of the first invocation of this method. This method will be called following an invocation of setUp.

3.3.5 CyclicSchedule

Declaration

```
@SCJAllowed
public class CyclicSchedule
```

Description

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

Constructor

```
@SCJAllowed
public CyclicSchedule(CyclicSchedule.Frame[] frames)
```

Construct a CyclicSchedule by copying the frames array into a private array within the same memory area as this newly constructed CyclicSchedule object. Under nor-

mal circumstances, the CyclicSchedule is constructed within the same MissionMemory area that holds the Level0Mission that is to be scheduled.

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 Level0Mission is idle.

Methods

3.3.6 CyclicSchedule.Frame

Declaration

@SCJAllowed
final public static class CyclicSchedule.Frame

Description

Constructor

@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.

Methods

3.3.7 Level0Mission

Declaration

@SCJAllowed
public abstract class Level0Mission **extends** Mission

Description

A Level 0 SCJ application is comprised of one or more Level0Mission. Each Level0Mission is implemented as a subclass of this abstract Level0Mission class.

Constructor

```
@SCJAllowed
public Level0Mission()
```

Constructor for a Level0Mission. Normally, application-specific code found within the application-defined subclass of MissionSequencer instantiates a new Level0Mission in the MissionMemory area that is dedicated to that Level0Mission. Upon entry into the constructor, this same MissionMemory area is the current allocation area.

Note that this class inherits missionMemorySize, initialize, requestTermination, terminationPending, requestSequenceTermination, sequenceTerminationPending, and cleanup methods from Mission.

Methods

```
@SCJAllowed
protected abstract CyclicSchedule getSchedule()
```

Return the CyclicSchedule for this Level0Mission, residing in the same scope as this Level0Mission object. Under normal circumstances, this method is only invoked from the SCJ infrastructure. The infrastructure invokes getSchedule after control returns from Mission.initialize and before the infrastructure "starts up" execution of the managedSchedulables instantiated and registered by the initialize method.

3.3.8 Level0MissionSequencer

Declaration

```
@SCJAllowed
public class Level0MissionSequencer extends MissionSequencer
```

Description

A Level0MissionSequencer differs from a MissionSequencer in that all of the Missions returned from getNextMission are instances of Level0Mission. This class inherits its requestSequenceTermination and sequenceTerminationPending from MissionSequencer.

Constructor

```
@SCJAllowed
@SCJRestricted({INITIALIZATION})
public Level0MissionSequencer(PriorityParameters priority,
                              StorageConfigurationParameters storage)
```

Construct a Level0MissionSequencer to run at the priority and with memory resources specified by its parameters.

Throws IllegalStateException if not invoked during initialization of a Level 2 Mission or during the infrastructure's invocation of an SCJ application's Safelet.getSequencer method.

Methods

```
@SCJAllowed
protected abstract Level0Mission getNextMission()
```

This method is called by the Level 0 infrastructure to select the initial Level0Mission to execute, and subsequently, each time one Level0Mission terminates, to determine the next Level0Mission to execute.

Prior to each invocation of getNextMission by the infrastructure, the infrastructure instantiates and enters a very large MissionMemory allocation area. The typical behavior is for getNextMission to return a Level0Mission object that resides in this MissionMemory area.

Returns the next Level0Mission to run, or null if no further Level0Missions are to run under the control of this Level0MissionSequencer.

3.3.9 SingleMissionSequencer

Declaration

```
@SCJAllowed
public class SingleMissionSequencer extends Level0MissionSequencer
```

Description

The SingleMissionSequencer class simplifies development of simple Level 0 applications comprised of a single Level0Mission that executes only once.

Constructor

```
@SCJAllowed
@BlockFree
@SCJRestricted({INITIALIZATION})
public SingleMissionSequencer(PriorityParameters priority,
                              StorageConfigurationParameters storage)
```

Construct a SingleMissionSequencer to run at the priority and with memory resources specified by its parameters.

Throws IllegalStateException if invoked during initialization of a Level 1 or Level 2 Mission.

Methods

```
@SCJAllowed
protected Level0Mission getNextMission()
```

On the first invocation, this returns the result of invoking the static Cyclet.getPrimordialMission method. On subsequent invocations, this returns null.

3.4 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. The sequence diagram below illustrates the interactions between the infrastructure and application code during the execution of a Level 1 application.

3.5 A Sample Level 0 Application

The following example illustrates how a simple Level 0 application could be written.

MyLevel0Mission

The starting point in Level 0 application is a subclass of Level0Mission, in this case we name it MyLevel0Mission. It must provide implementations of the abstract methods declared in its parent class.

```
@SCJAllowed(members=true)
class MyLevel0Mission extends Level0Mission {
  PeriodicEventHandler[] pehs = null;

  public long missionMemorySize() { return 10000L; }

  /**
   * Create the event handlers that will be used in the application.
   * These periodic event handlers are recorded by the infrastructure
   * as they are created and do not need to be explicitly registered.
   */
  public synchronized void initialize() {
    pehs = new PeriodicEventHandler[3];

    pehs[0] = new MyPEH("A",new RelativeTime(0,0),new RelativeTime(500,0));
    pehs[1] = new MyPEH("B",new RelativeTime(0,0),new RelativeTime(1000,0));
    pehs[2] = new MyPEH("C",new RelativeTime(0,0),new RelativeTime(500,0));

    pehs[0].register();
    pehs[1].register();
    pehs[2].register();
  }
```

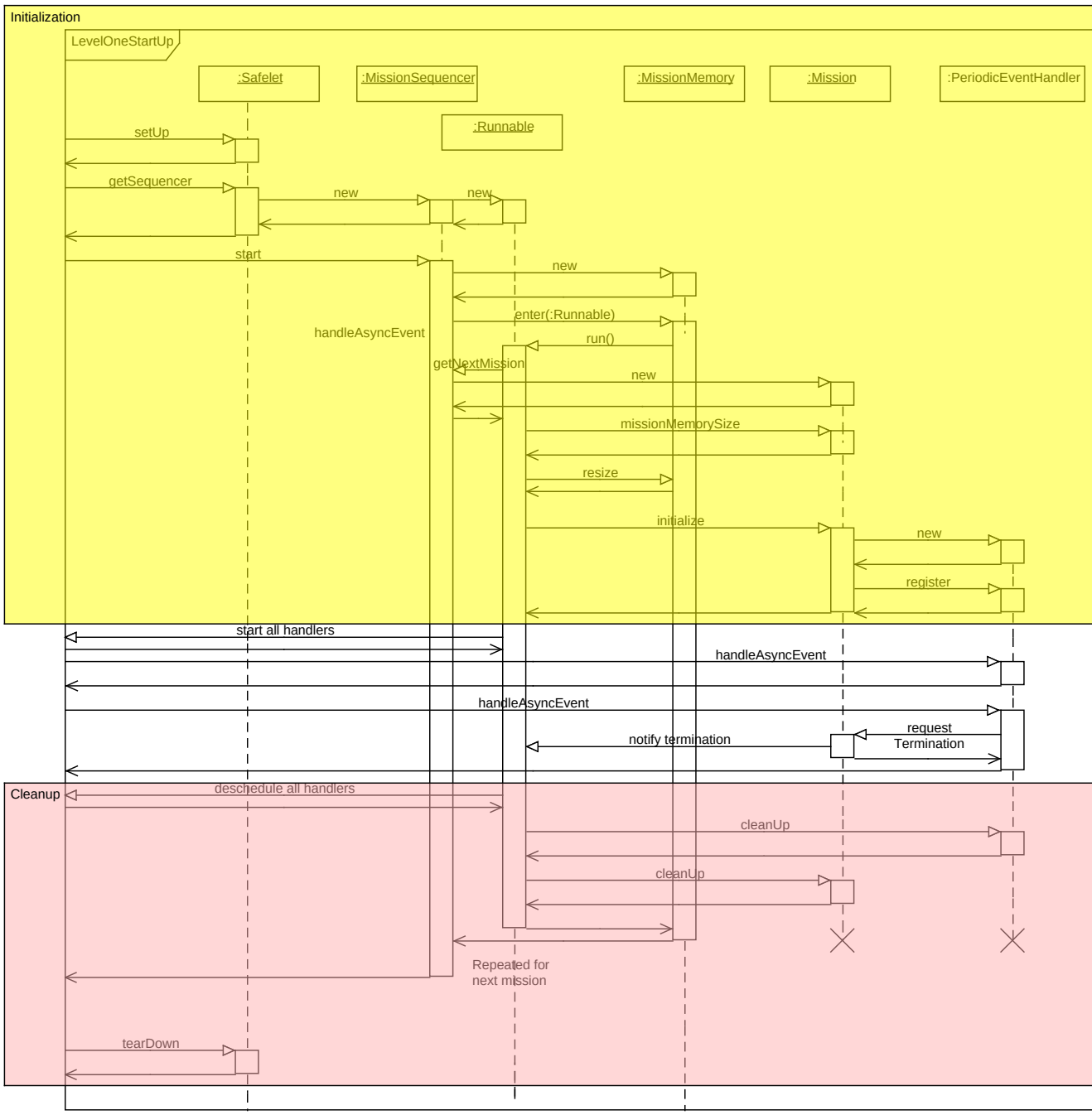

Figure 3.2: Sequence diagram for a level 1 application

```
  // note that getSchedule may be invoked from a different thread than
  // initialize, so access to pehs needs to be synchronized. Or, do we
  // want to spec that this is always the same thread?
  protected synchronized CyclicSchedule getSchedule() {
    return VendorCyclicSchedule.generate(this, pehs);
  }
}
```

MyPEH

The application logic is defined in subclasses of PeriodicEventHandler. For this sample application we define a single subclass called MyPEH which has a handleAsyncEvent method that simply increments a counter.

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

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

  MyPEH(String nm, RelativeTime start, RelativeTime period) {
     super(new PriorityParameters(priority),
           new PeriodicParameters(start, period), mSize);
  }

  public void handleAsyncEvent() {
     ++eventCounter;
  }
}
```

The schedule provided is intended to be a tool-generated schedule.

VendorCyclicSchedule

A sample implementation of a cyclic schedule, generated by a vendor-specific tool, is shown next. In this example, the generated schedule is for an application that has three asynchronous event handlers that are 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.

```
@SCJAllowed(members=true)
class VendorCyclicSchedule {

  static Level0Mission 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.
   */
  static CyclicSchedule generate(PeriodicEventHandler[] peh,
                                 Level0Mission m) {
    if (m == cache_key)
      return cache_schedule;
    else {
      // ideally, the following five allocations would be taken from
      // a PrivateMemory as they are not needed following return from
      // this method, but it's too much work to write it that way, so
      // we'll just leave a bit of "pollution" in the MissionMemory.
      CyclicSchedule.Frame frames[] = new CyclicSchedule.Frame[2];
      PeriodicEventHandlers frame1_handlers[] = new PeriodicEventHandlers[3];
      PeriodicEventHandlers frame2_handlers[] = new PeriodicEventHandlers[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;
    }
}
```

Configuration of the Level 0 Application

Vendor-specific tools are used to configure a Java application. The parameters that must be specified to properly configure execution of this sample application are described below:

- The class that represents the primordial Level0Mission that comprises this simple application. It is assumed that the class has a no-argument constructor. If

a Level 0 application's Level0Mission constructor requires arguments, an alternative approach configuration option is available, as described in the next section.

- The priority at which the application's Level0MissionSequencer is intended to run.
- The value of the StorageConfigurationParameters for the application's Level0MissionSequencer thread. Note that the resources budgeted for the Level0MissionSequencer thread must be large enough to be divided into the memory for each of the Level0Missions represented by the Level0MissionSequencer. Each Level0Mission requires resources for the MissionMemory and it needs resources to satisfy the StorageConfigurationParameters demand of each managedSchedulable associated with the Level0Mission.

Assume, for illustration, that configuration parameters are supplied to a build-time tool. One possible implementation technique would arrange for automatic generation of a package-access class to represent the specified configuration parameters. Suppose that the system developer specifies the following values for the configuration parameters.

- The primordial Level0Mission is MyLevel0Mission.
- The desired priority for execution of the Level0MissionSequencer is 18.
- The desired StorageConfigurationParameters for the Level0MissionSequencer is (totalBackingStore = 100000), (nativeStackSize = 10000), (javaStackSize = 5000), (messageLength = 80), and (stackTraceLength = 512).

At configuration time, a tool outputs the following class definition:

```
class Configuration {

  static Level0Sequencer getSequencer() {
    return new SingleMissionSequencer(sequencerPriority(), sequencerStorage());
  }

  static Level0Mission primordialMission() {
    return new MyLevel0Mission();
  }

  static PriorityParameters sequencerPriority() {
    return new javax.realtime.PriorityParameters(18);
  }

  static StorageParameters sequencerStorage() {
    return new StorageParameters(100000, 10000, 5000, 80, 512);
  }
}
```

The compatible implementation of Cyclet.getSequencer might consist of the following.

```
@SCJAllowed
@SCJRestricted({INITIALIZATION})
public Level0MissionSequencer getSequencer()
{
  return new SingleMissionSequencer(Configuration.sequencerPriority(),
                                    Configuration.sequencerStorage());
}
```

The compatible implementation of Cyclet.getPrimordialMission might consist of the code that follows:

```
  public static Level0Mission getPrimordialMission() {
    return Configuration.primordialMission();
  }
```

3.6 A Slightly More Complex Level 0 Application

This sample application implements similar functionality to the preceding example. It uses an application-defined Level0MissionSequencer instead of SingleMissionSequencer. Add to the code described above the following classes.

```
@SCJAllowed(members=true)
class MyLevel0App extends Cyclet {

  public MyLevel0App() {
    super();
  }

  @SCJRestricted({INITIALIZATION})
  public Leve0MissionSequencer getSequencer() {
    PriorityParameters p = new PriorityParameters(18);
    StorageParameters s = new StorageParameters(100000, 10000, 5000, 80, 512);
    return new MyLevel0Sequencer(p, s);
  }
}

@SCJAllowed(members=true)
class MyLevel0Sequencer extends Level0MissionSequencer {

  public MyLevel0Sequencer(PriorityParameters p, StorageParameters s) {
    super(p, s);
  }
```

```
  protected Level0Mission getNextMission() {
    if (providedInitialMission) {
      return null;
    }
    else {
      return new MyLevel0Mission();
    }
  }
}
```

This application runs the same as the preceding example except that if MyLevel0Mission terminates, MyLevel0Sequencer causes the mission to restart instead of ending the application.

When application developers provide their own implementation of Cyclet and Level0MissionSequencer, configuration of the application does not have to specify the primordial mission, nor the priority or StorageConfigurationParameters of the Level0MissionSequencer, because these values are hard-coded into the implementation of the application-specific Cyclet subclass. It is however necessary in this case for the application developer to specify at configuration time the subclass of Cyclet that represents the application.

3.7 Level 2 Example

The following example illustrates how a simple Level 2 application could be written with nested missions.

3.7.1 MyLevel2App.java

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

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

  public MissionSequencer getSequencer() {
    StorageConfigurationParameters sp =
      new StorageConfigurationParameters(100000L, 1000L, 1000L);
    return new MainMissionSequencer(new PriorityParameters(PRIORITY), sp);
  }

  public void setup() {}
  public void teardown() {}

}
```

3.7.2 MainMissionSequencer.java

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

  private boolean initialized, finalized;

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

  protected Mission getNextMission() {
    if (finalized)
      return false;
    else if (initialized) {
      finalized = true;
      return new CleanupMission();
    }
    else {
      initialized = true;
      return new PrimaryMission();
    }
  }
}
```

3.7.3 PrimaryMission.java

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

  public void initialize() {
    PriorityParameters pp = new PriorityParameters(PRIORITY);
    StorageConfigurationParameters sp =
      new StorageConfigurationParameters(100000L, 1000L, 1000L);
    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();
  }

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

3.7.4 CleanupMission.java

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

  public void initialize() {
    PriorityParameters pp = new PriorityParameters(PRIORITY);
    StorageConfigurationParameters sp =
      new StorageConfigurationParameters(100000L, 1000L, 1000L);
    MyCleanupThread t = new MyCleanupThread(pp, sp);
  }

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

3.7.5 SubMissionSequencer.java

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

  SubMissionSequencer(PriorityParameters priorityParameters,
                      StorageConfigurationParameters 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();
    }
  }
}
```

3.7.6 StageOneMission.java

```
@SCJAllowed(members=true, value=LEVEL_2)
public class StageOneMission extends Mission {
  public void initialize() {
    (new MyPeriodicEventHandler(“stage1.eh1”,
                                new RelativeTime(0, 0),
```

```
                                    new RelativeTime(1000, 0))).register();
  }
  public long missionMemorySize() { return 100;}
}
```

3.7.7 StageTwoMission.java

```
@SCJAllowed(members=true, value=LEVEL_2)
public class StageTwoMission extends Mission {
  public void initialize() {
    (new MyPeriodicEventHandler("stage2.eh1",
                                    new RelativeTime(0, 0),
                                    new RelativeTime(500, 0))).register();
  }
  public long missionMemorySize() { return 100; }
}
```

3.7.8 MyPeriodicEventHandler.java

```
@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), _memSize);
  }

  public void handleAsyncEvent() {
    ++_eventCounter;
  }

  public void cleanup() {}
}
```

3.7.9 MyCleanupThread.java

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

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

  public void run() {
```

```
    cleanupThis();
    cleanupThat();
  }

  void cleanupThis() {
    // code not shown
  }

  void cleanupThat() {
    // code not shown
  }

}
```

Chapter 4

Concurrency and Scheduling Models

A schedulable object executes in response to 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—usually event-triggered.

Communication between schedulable objects is supported using shared variables and therefore requires synchronization and priority inversion management protocols. On multi-processor platforms[2], the assumption is that all processors can access all shared data, although not necessarily with uniform access times.

SCJ specifies the constraints placed on the RTSJ concurrency and scheduling models. It supports this constrained model by defining its own classes, all of which are implementable in the RTSJ. SCJ also requires 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.

SCJ extends the RTSJ to support

- storage parameters – this allows, for example, the Java stack size of a schedulable object to be specified;

[1] Please refer to the RTSJ specification [**?**] for a rigorous definition.

[2] The term *processor* is used in this chapter to indicate a logical processing element that is capable of physically executing a single thread of control at any point in time. Hence, multicore platforms have multiple processors, platforms that support hyperthreading also have more than one processor. It is assumed that all processors are capable of executing the same instruction sets. Hence virtual and logical processor are supported.

- scheduling allocation domains – each of these defines a set of processors on which schedulable objects executions can be constrained; they are implemented in terms of RTSJ AffinitySets; A scheduling allocation domain is one or more processors on which schedulable objects are globally scheduled. A processor cannot exist in more than one scheduling allocation domain.
- missions – all schedulable objects execute in the context and confines of a mission (see Chapter 2).

4.1 Semantics and Requirements

The goals for the SCJ concurrency model are 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 support traditional cyclic scheduling in order to support the migration from current sequential systems to concurrent systems.

The following requirements apply across all conformance levels.

- The number of processors allocated to Java platform shall be fixed and non changing.
- The number of scheduling allocation domains shall be fixed.
- Only no-heap and non-daemon RTSJ schedulable objects shall be supported (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 as considered to be aperiodic. There is no support for CPU-time monitoring and processing group parameters.
- The thread of control provided by the real-time virtual machine that executes the initialization phase is a bound aperiodic asynchronous event handler. This is a departure from standard RTSJ semantics where the main method is executed by a Java thread.
- The default ceiling for locks used by the application and the infrastructure shall be set to javax.safetycfritical.PriorityScheduler.instance().getMaxPriority() (that is, the maximum value for local ceilings – see Section 4.6.6)
- A preempted schedulable object shall be 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.

The following lists the main requirements on application designers:

1. Shared objects are represented by classes with synchronized methods. No use of the synchronized statement is allowed.

2. Use of the Object.wait in level 2 code should only name this.

3. Nested calls from one synchronized method to another are allowed. The ceiling priority associated with a nested synchronized method call must be greater than or equal to the ceiling priority associated with the outer call.

4. At all levels synchronized code is not allowed to self suspend (for example as a result of an IO request or the sleep method call). An IllegalMonitorStateException is thrown if this constraint is violated. Requesting a lock (via the synchronized method) is not considered self-suspension.

4.2 Level Considerations

The following defines those semantics that apply at different levels.

4.2.1 Level 0

- Only periodic bound asynchronous event handlers (class PeriodicEventHandler) shall be supported.
- There are a fixed number of implementation-predefined affinity sets, each of which contains only a single processor. No dynamic creation of affinity sets is allowed.
- Calls to Object.wait, Object.notify and Object.notifyAll are not allowed.
- Only one thread of control shall be provided by the real-time virtual machine to execute handlers. The handlers shall be executed non preemptively. A table-driven approach is acceptable, with the schedule being computed statically off-line in an implementation-defined manner.
- No locking is required for the implementation of synchronized code, but it is recommended that the applications use synchronized methods to support portability of code between levels.
- There is no deadline miss detection facility.

4.2.2 Level 1

- Aperiodic asynchronous event handlers (class AperiodicEventHandler) shall be supported.
- Each event handler shall be permanently bound to its own implementation-defined thread of control, and each thread of control shall only be bound to a single event handler.

- There are a fixed number of implementation-predefined affinity sets, each of which contains only a single processor. No dynamic creation of affinity sets is allowed.
- Calls to Object.wait, Object.notify and Object.notifyAll are not allowed.
- Full preemptive scheduling shall be supported. The only scheduler is the default RTSJ preemptive priority-based scheduler with at least 28 (software and hardware) priorities and priority ceiling emulation (although if portability is a main concern, no more than 28 priorities should be used). There is no support for changing base priorities.
- Deadline miss detection shall be supported. An implementation is required to document the granularity at which missed deadlines are detected (see Section 4.7.5). The deadline miss shall be signalled no earlier that the deadline of the associated managed event handler.

4.2.3 Level 2

- NoHeapRealtimeThreads shall be supported but be managed (the ManagedThread class).
- There are a fixed number of implementation-predefined affinity sets. Each affinity set may contain one or more processors. However, no processor can appear in more than one affinity set.
- Dynamic creation of affinity sets is allowed during the mission initialization phase, but each set shall only contain a single processor.
- Calls to Object.wait, Object.notify and Object.notifyAll are allowed. However, calling Object.wait from nested synchronization code is illegal.

4.3 The Parameter Classes

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

- StorageParameters — these allow, for example, the stack size of a schedulable object to be specified.
- The ReleaseParameters class hierarchy — these allow the release characteristics of a schedulable object to be specified, for example whether it is periodic or aperiodic.
- The SchedulingParameters class hierarchy — these allow the priorities of the schedulable objects to be set.

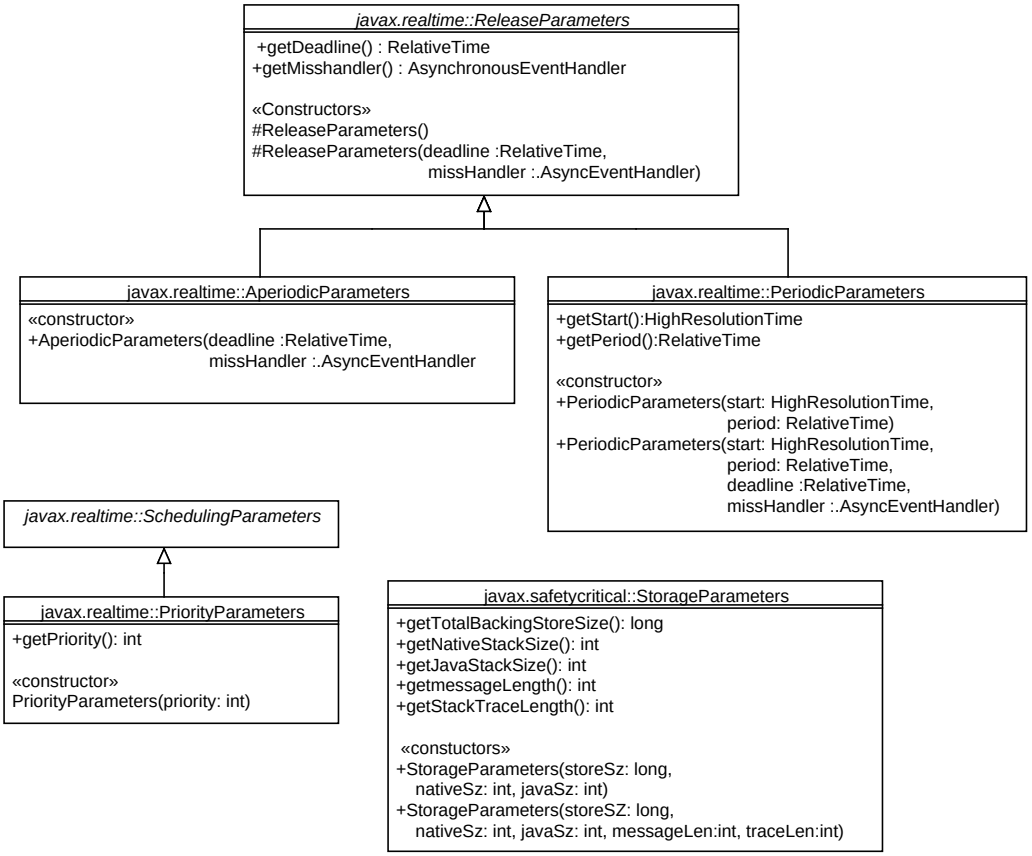

Figure 4.1: Parameter classes

4.3.1 Class java.safetycritical.StorageParameters

Declaration

@SCJAllowed
public class StorageParameters

Description

Each schedulable object has several associated types of storage. As well as its Java run-time execution stack, there is also a native method stack (if this memory is distinct from the run-time stack). In addition, each schedulable object has: a backing store that is used for any scoped memory areas it may create and a number of bytes dedicated to the message associated with this Schedulable object's ThrowBoundaryError exception plus all the method names/identifiers in the stack backtrace.

This class allows the programmer to set the sizes of these memory stores only at construction time (the objects are immutable). It is assumed that these sizes are

obtained from vendor-specific tools.

Constructors

```
@SCJAllowed
public StorageParameters(long storeSz, int nativeSz, int javaSz)
```

Stack sizes for schedulable objects and sequencers passed as parameter to the constructor of mission sequencers and schedulable objects.

Parameter storeSz is the size of the backing store reservation in bytes.

Parameter nativeSz is the size of the native stack in bytes.

Parameter javaSz is the size of the Java stack in bytes.

The default messageLen and traceLen values are set during the configuration of the virtual machine.

Throws IllegalArgumentException if one or more of the parameters are less than zero.

```
@SCJAllowed
public StorageParameters(long storeSz, int nativeSz, int javaSz,
                                        int messageLen, int traceLen)
```

Stack sizes for schedulable objects and sequencers passed as parameter to the constructor of mission sequencers and schedulable objects.

Parameter storeSz is the size of the backing store reservation in bytes.

Parameter nativeZs is the size of the native stack in bytes.

Parameter javaSz is the size of the Java stack in bytes.

Parameter messageLen is the space in bytes dedicated to message associated with this Schedulable object's ThrowBoundaryError exception plus all the method names and identifiers in the stack backtrace.

Parameter traceLen is the number of bytes for theStackTraceElement array dedicated to stack backtrace associated with this Schedulable object's ThrowBoundaryError exception.

Throws IllegalArgumentException if one or more of the parameters are less than zero.

Methods

```
@SCJAllowed
public long getTotalBackingStoreSize()
```

Returns the size of the total backing store available for scoped memory areas created by the assocated Schedulable object.

```
@SCJAllowed
public int getNativeStackSize()
```

Returns the size of native method stack available to the assocated Schedulable object.

@SCJAllowed
public int getJavaStackSize()

Returns the size of Java method stack available to the assocated Schedulable object.

@SCJAllowed
public int getMessageLength()

Returns the length of the message buffer in bytes.

@SCJAllowed
public int getStackTraceLength()

Returns the length of the stack trace buffer in bytes.

4.3.2 Class javax.realtime.ReleaseParameters

Declaration

@SCJAllowed
public abstract class ReleaseParameters **implements** Cloneable

Description

All analysis of safety critical software is performed off line. 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 failure hypothesis is that deadlines should not be missed. However, to facilitate fault-tolerant applications, SCJ does support a deadline miss detection facility at Levels 1 and 2. SCJ provides no direct mechanisms for coping with cost overruns. These decisions are reflected in the ReleaseParameters class and its subclasses.

The ReleaseParameters class is restricted so that it allows the parameters to be set and queried but not changes. 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
protected ReleaseParamters()

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

@SCJAllowed(LEVEL_1)
protected ReleaseParamters(RelativeTime deadline, AsyncEventHandler missHandler)

Construct an object which has deadline checking facility.

Parameter deadline is the deadline to be checked.

Parameter missHandler is the event handler to be released when the deadline miss has been detected.

Methods

```
@SCJAllowed(LEVEL_1)
public Object clone()
```

```
@SCJAllowed(LEVEL_1)
public RelativeTime getDeadline()
```

Returns the deadline set at construction time.

```
@SCJAllowed(LEVEL_1)
public AsyncEventHandler getDeadlineMissHandler()
```

Returns the handler set at construction time. Returns null if there is no deadline miss detection facility.

4.3.3 Class java.realtime.PeriodicParameters

Declaration

```
@SCJAllowed
public class PeriodicParameters extends ReleaseParameters
```

Description

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

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
public PeriodicParameters(HighResolutionTime start, RelativeTime period)
```

Construct a new object within the current memory area. The default deadline is the same value as period.

```
@SCJAllowed(LEVEL_1)
public PeriodicParameters(HighResolutionTime start, RelativeTime period
                          RelativeTime deadline, AsyncEventHandler missHandler)
```

Construct a new object within the current memory area.

Methods

@SCJAllowed
public HighResolutionTime getStart()
Returns a newly created object whose time is the same as the start time passed at construction.

@SCJAllowed
public RelativeTime getPeriod()
Returns a newly created object whose time is the same as the period time passed at construction.

4.3.4 Class javax.realtime.AperiodicParameters

Declaration

@SCJAllowed(LEVEL_1)
public class AperiodicParameters **extends** 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. However there the arrival time queue length is assumed to be 1, and the overflow behaviour is throw an exception on firing the associated event.

Constructors

@SCJAllowed(LEVEL_1)
public AperiodicParameters(RelativeTime deadline, AsyncEventHandler missHandler)
Construct a new object within the current memory area.

4.3.5 Class javax.realtime.SchedulingParameters

Declaration

@SCJAllowed
public abstract class SchedulingParameters
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.

There is no ImportanceParameters subclass in SCJ.

4.3.6 Class javax.realtime.PriorityParameters

Declaration

```
@SCJAllowed
public class PriorityParameters extends SchedulingParameters
```

Description

The class is restricted 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 4.6.6). 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.

Constructor

```
@SCJAllowed
public PriorityParameters(int priority)
```

Create an object within the current allocation context with the value priority.

Methods

```
@SCJAllowed
public int getPriority()
```

Returns the integer priority value that was passed at construction time.

4.4 Asynchronous Events and their Handlers

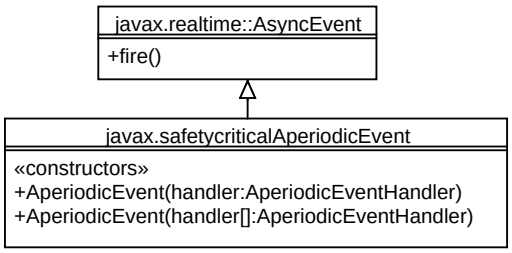

Figure 4.2: Async Event Classes

The event based programming paradigm in SCJ is implemented using the RTSJ asynchronous event handling mechanisms (see Figures 4.2 and 4.3). The types of events and event handlers have very constrained use cases in SCJ. Consequently

new subclasses are defined to support these. Direct use of the RTSJ classes by the application is therefore disallowed.

All asynchronous-events related classes in SCJ are rooted the AsyncEvent class.

SCJ supports two types of events and their handlers:

- periodic events — the only form of periodic events that is supported are those that are generated by the passage of time. SCJ does not provide visibility to the infrastructure code that generates these timing events. Hence, it only provides a PeriodicEventHandler class.
- aperiodic events and their handlers — both software generated events and external events are support. The latter is discussed in the context of external events in section 5.

4.4.1 Class javax.realtime.AsyncEvent

Declaration

```
@SCJAllowed(LEVEL_1)
public class AsyncEvent
```

Description

In SCJ only aperiodic event are available at the application level. Hence, constructors are hidden from public view. Handlers must be attached when events are created. Consequently the related methods have been removed. There is also no support for binding to external happenings using this class.

Methods

```
@SCJAllowed(LEVEL_1)
public void fire()
```

Fire this event, i.e., releases all the handlers that have been added to this event.

4.4.2 Class javax.safetycritical.AperiodicEvent

Declaration

```
@SCJAllowed(LEVEL_1)
public class AperiodicEvent extends AsyncEvent
```

Description

A class of events that enables code to release the execution of AperiodicEventHandlers. The event is software-triggered. As there is no support for minimal inter-arrival time monitoring, there is no support for sporadic events and their handlers.

Constructors

@SCJAllowed(LEVEL_1)
@SCJRestricted(INITIALIZATION)
public AperiodicEvent(AperiodicEventHandler handler)

Constructor for an aperiodic event that is linked to a given handler.

Throws IllegalArgumentException if handler is null.

@SCJAllowed(LEVEL_1)
@SCJRestricted(INITIALIZATION)
public AperiodicEvent(AperiodicEventHandler[] handlers)

Constructor for an aperiodic event that is linked to multiple handlers.

Throws IllegalArgumentException if handlers is null.

4.4.3 Class javax.realtime.Schedulable

Declaration

@SCJAllowed
public interface Schedulable **extends** 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. In SCJ, it provides no extra functionality over the Runnable interface.

4.4.4 Class javax.safetycritical.ManagedSchedulable

Declaration

@SCJAllowed
public interface ManagedSchedulable **extends** Schedulable

Description

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

Methods

@SCJAllowed
public void register()

Register this schedulable object with the current mission.

@SCJAllowed
public void cleanUp()

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

4.4.5 Class javax.realtime.AsyncEventHandler

Declaration

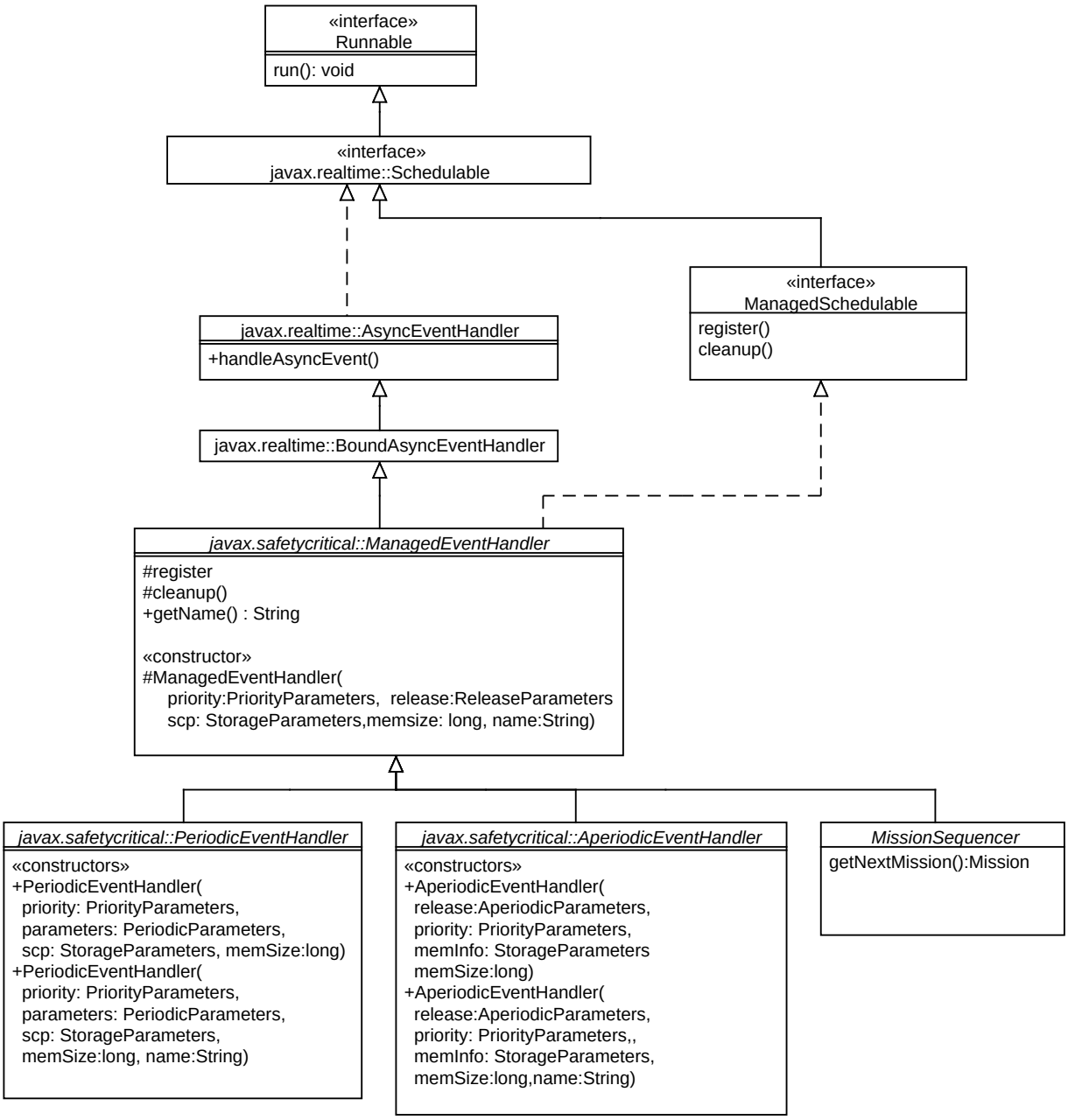

Figure 4.3: Handler classes

```
@SCJAllowed
public class AsyncEventHandler implements Schedulable
```

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 AsyncEventHandler constructors are hidden from public view in the SCJ specification.

Methods

```
@SCJAllowed
public void handleAsyncEvent()
```

This is overridden by the application to provide the handling code.

4.4.6 Class javax.realtime.BoundAsyncEventHandler

Declaration

```
@SCJAllowed
public class BoundAsyncEventHandler extends AsyncEventHandler
```

Description

The BoundAsyncEventHandler class is not used directly by the application. Hence none of its methods or constructors are publicly available.

4.4.7 Class javax.safetycritical.ManagedEventHandler

Declaration

```
@SCJAllowed
public abstract class ManagedEventHandler extends BoundAsyncEventHandler
                                          implements ManagedSchedulable
```

Description

In SCJ, all handlers must be known by the mission manager, hence applications use classes that are based on the ManagedEventHandler class hierarchy. This class hierarchy allows a mission to keep track of all the handlers that are created during the initialization phase.

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

Constructor

```
@SCJAllowed
@SCJRestricted(INITIALIZATION)
protected ManagedEventHandler(PriorityParameters priority,
                              ReleaseParameters release,
                              StorageParameters storage,
                              long memSize, String name)
```

Constructor to create an event handler.

Parameter priority specifies the priority parameters for this periodic event handler.

Parameter release specifies the non-null release parameters.

Parameter storage describes the organization of memory dedicated to execution of the underlying thread.

Parameter memSize is the size in bytes of the private scoped memory area to be used for the execution of this event handler. 0 for an empty memory area.

Throws IllegalArgumentException if either priority or release is null or if memSize is negative.

Methods

```
@SCJAllowed
@SCJRestricted(CLEANUP)
protected void cleanup()
```

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

```
@SCJAllowed
public String getName()
```

Returns a string name for this handler, including its priority.

4.4.8 Class javax.safetycritical.PeriodicEventHandler

Declaration

```
@SCJAllowed
public abstract class PeriodicEventHandler extends ManagedEventHandler
```

Description

This class permits the automatic periodic execution of code. It is automatically bound to an infrastructure periodic timer in the constructor. This class is abstract, non-abstract sub-classes must implement the methods handleAsyncEvent and cleanup.

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.

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(INITIALIZATION)
public PeriodicEventHandler(PriorityParameters priority, PeriodicParameters release,
                            StorageParameters storage, long memSize)
```

Constructor to create a periodic event handler.

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

Parameter release specifies the periodic release parameters, in particular the start time, period, deadline 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.

Parameter storage parameter describes the organization of memory dedicated to execution of the underlying thread.

Parameter memSize is the size in bytes of the memory area to be used for the execution of this event handler. 0 for an empty memory area. Must not be negative.

Throws IllegalArgumentException if either priority, release is null, or if memSize is negative.

```
@SCJAllowed(LEVEL_1)
@SCJRestricted(INITIALIZATION)
public PeriodicEventHandler(PriorityParameters priority, PeriodicParameters release,
                            StorageParameters storage, long memSize, String name)
```

Constructor to create a periodic event handler.

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

Parameter release specifies the periodic release parameters, in particular the start time, period, deadline 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.

Parameter storage parameter describes the organization of memory dedicated to execution of the underlying thread.

Parameter memSize is the size in bytes of the private scoped memory area to be used for the execution of this event handler. 0 for an empty memory area. Must not be negative.

Throws IllegalArgumentException if either priority, release is null, or if memSize is negative.

4.4.9 Class javax.safetycritical.AperiodicEventHandler

Declaration

```
@SCJAllowed(LEVEL_1)
public abstract class AperiodicEventHandler extends 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 handleAsyncEvent and override the default cleanup methods. 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(LEVEL_1)
@SCJRestricted(INITIALIZATION)
public AperiodicEventHandler(PriorityParameters priority,
                             StorageParameters storage, long memSize)
```

Constructor to create an aperiodic event handler.

Parameter priority specifies the priority parameters for this aperiodic event handler; it must not be null.

Parameter storage parameter describes the organization of memory dedicated to execution of the underlying thread.

Parameter memSize is the size in bytes of the memory area to be used for the execution of this event handler. 0 for an empty memory area. Must not be negative.

Throws IllegalArgumentException if priority or storage is null, or if memSize is negative.

```
@SCJAllowed(LEVEL_1)
@SCJRestricted(INITIALIZATION)
public AperiodicEventHandler(PriorityParameters priority,
                             StorageParameters storage, long memSize, String name)
```

Constructor to create an aperiodic event handler.

Parameter priority specifies the priority parameters for this aperiodic event handler; it must not be null.

Parameter storage parameter describes the organization of memory dedicated to execution of the underlying thread.

Parameter memSize is the size in bytes of the memory area to be used for the execution of this event handler. 0 for an empty memory area. Must not be negative.

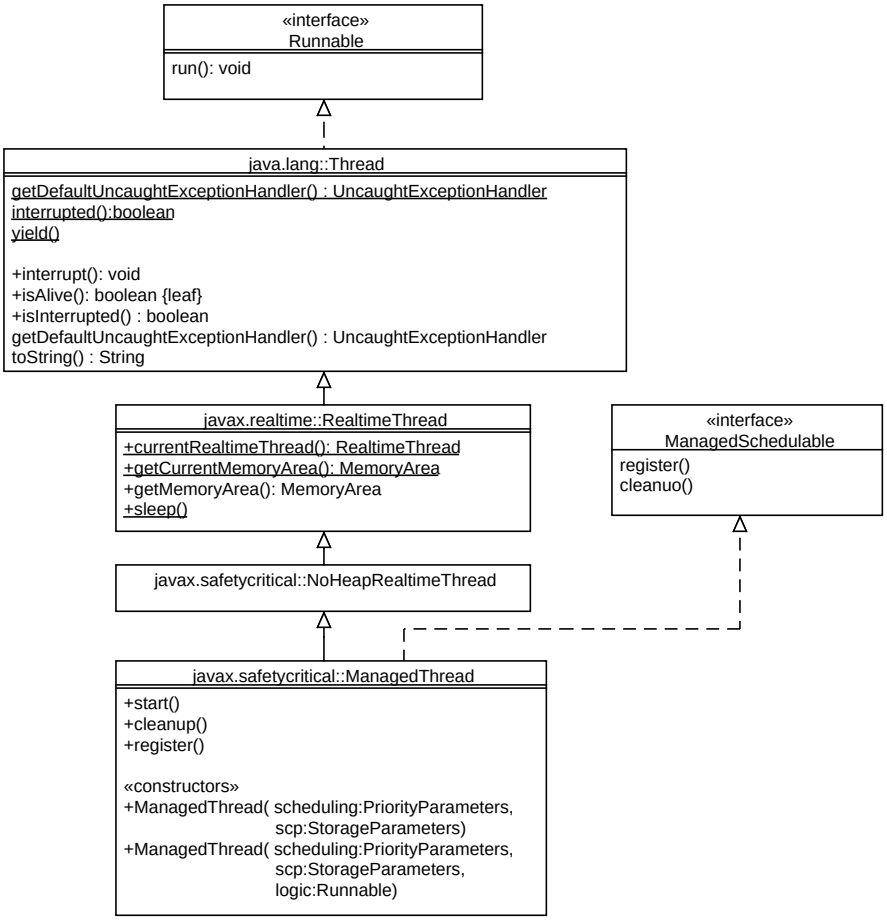

Figure 4.4: Thread classes

Throws IllegalArgumentException if priority or storage is null, or if memSize is negative.

4.5 Threads and Real-Time Threads

In keeping with the approach outlined above for events and their handlers, the threads APIs are equally simple. They are shown in Figure 4.4.

4.5.1 Class java.lang.Thread

Declaration

@SCJAllowed
public class Thread **implements** Runnable

Description

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

Static Methods

@SCJAllowed(LEVEL_2)
public static UncaughtExceptionHandler getDefaultUncaughtExceptionHandler()

Returns the default handler for uncaught exceptions.

@SCJAllowed(LEVEL_2)
public static boolean interrupted()

Returns true if the current thread has been interrupted.

@SCJAllowed(LEVEL_2)
public static void yield()

Causes the currently executing thread object to temporarily pause and allow other threads to execute.

Methods

@SCJAllowed(LEVEL_2)
public void interrupt()

Interrupts this thread.

@SCJAllowed(LEVEL_2)
public final boolean isAlive()

Returns true if this thread is alive. False otherwize.

@SCJAllowed(LEVEL_2)
public boolean isInterrupted()

Returns true if this thread has been interrupted. False otherwize.

@SCJAllowed(LEVEL_2)
public UncaughtExceptionHandler getUncaughtExceptionHandler()

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

@SCJAllowed
public String toString()

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

4.5.2 Class java.lang.Thread.UncaughtExceptionHandler

Declaration

@SCJAllowed(LEVEL_2)
public static interface UncaughtExceptionHandler

Interface for handlers invoked when a Thread abruptly terminates due to an uncaught exception.

Methods

void uncaughtException(Thread t, Throwable e)

Method invoked when the given thread terminates due to the given uncaught exception.

4.5.3 Class javax.realtime.RealtimeThread

Declaration

@SCJAllowed(LEVEL_1)
public class RealtimeThread **extends** Thread **implements** Schedulable

Description

Real-time threads cannot be directly created by the 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.

Static Methods

@SCJAllowed(LEVEL_2)
public static RealtimeThread currentRealtimeThread()

Returns a reference to the RealtimeThread calling this method.

@SCJAllowed(LEVEL_1)
public static MemoryArea getCurrentMemoryArea()

Returns a reference to the current allocation context.

@SCJAllowed(LEVEL_2)
@SCJRestricted{MAY_BLOCK)
public static void sleep(HighResolutionTime time) **throws** 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.

Methods

```
@SCJAllowed(LEVEL_2)
public MemoryArea getMemoryArea()
```

Returns a reference to the allocation context represented by this.

4.5.4 Class javax.realtime.NoHeapRealtimeThread

Declaration

```
@SCJAllowed(LEVEL_2)
public class NoHeapRealtimeThread extends RealtimeThread
```

Description

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

4.5.5 Class javax.safetycritical.ManagedThread

Declaration

```
@SCJAllowed(LEVEL_2)
public class ManagedThread extends RealtimeThread
                           implements ManagedSchedulable
```

Description

This class allows a mission to keep track of all the no-heap real-time threads that are created during the initialization phase.

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

Constructors

```
@SCJAllowed(LEVEL_2)
@SCJRestricted(INITIALIZATION)
public ManagedThread(PriorityParameters priority,
                     StorageParameters storage,
                     long memsize)
```

Creates a thread that is managed by the enclosing mission.

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

Parameter storage parameter describes the organization of memory dedicated to execution of this managed thread.

Parameter memSize is the size in bytes of the memory area to be used for the execution of this managed thread. 0 for an empty memory area. Must not be negative.

Throws IllegalArgumentException if priority, parameters or if memSize is negative.

```
@SCJAllowed(LEVEL_2)
@SCJRestricted(INITIALIZATION)
public ManagedThread(PriorityParameters priority,
                     StorageParameters storage,
                     long memsize
                     Runnable logic)
```

Creates a thread that is managed by the enclosing mission.

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

Parameter storage parameter describes the organization of memory dedicated to execution of this managed thread.

Parameter memSize is the size in bytes of the memory area to be used for the execution of this managed thread. 0 for an empty memory area. Must not be negative.

Parameter code is the code for this managed thread.

Methods

```
@SCJAllowed(LEVEL_2)
@SCJRestricted(INITIALIZATION)
public void start()
```

Start this managed thread.

```
@SCJAllowed
public void cleanUp()
```

Execute any clean up code associated with this managed thread.

```
@SCJAllowed
public void register()
```

Register this managed thread.

4.6 Scheduling and Related Activities

Level 0 applications are assumed to be scheduled by a cyclic executive where the schedule is created by static analysis tools offline. Level 1 and 2 applications are assumed to be scheduled by the RTSJ base level priority scheduler.

4.6.1 Class java.safetycritical.CyclicExecutive

Declaration

```
@SCJAllowed
public abstract class CyclicExecutive extends Mission implements Safelet
```

Level 0 applications are assumed to be scheduled by a cyclic executive where the schedule is created by static analysis tools offline.

Constructors

```
@SCJAllowed
public CyclicExecutive()
```

Level 0 Applications need to extend CyclicExecutive and define a getSchedule method.

Methods

```
@SCJAllowed
public abstract CyclicSchedule getSchedule(PeriodicEventHandler[] peh);
```

Returns the schedule to be used by the application. This will typically be tooling-generated.

```
@SCJAllowed
public MissionSequencer getSequencer()
```

Returns The sequencer to be used for the Level 0 application. By default this is a SingleMissionSequencer, although this method can be overridden by the application if an alternative sequencer is desired.

4.6.2 Class javax.safetycritical.CyclicSchedule

Declaration

```
@SCJAllowed
public class CyclicSchedule
```

Description

At Level 0, a cyclic schedule is represented by this class. There is one schedule for each processor in the system. The current class assumes a single processor.

Constructor

@SCJAllowed
public CyclicSchedule(CyclicSchedule.Frame[] frames)

Constructs a cyclic schedule. All the frames must be based on the same clock.

Methods

@SCJAllowed
public RelativeTime getCycleDuration()

Returns a newly allocated RelativeTime object, taken from the current memory area of the caller. This is the sum of the duration of all the frames' durations.

@SCJAllowed
protected Frame[] getFrames()

Returns an array that will contain references to the same Frame objects that are used internally by the infrastructure.

4.6.3 Class javax.safetycritical.CyclicSchedule.Frame

Declaration

@SCJAllowed
final public static class Frame

Constructor

@SCJAllowed
public Frame(RelativeTime duration, PeriodicEventHandler[] handlers)

Allocates private copies of the handlers 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.

Methods

@SCJAllowed
public RelativeTime getDuration()

Returns a reference to the internal representation of the frame duration.

@SCJAllowed
public PeriodicEventHandler[] getHandlers()

Returns an array allocated in the memory area of the caller. This array holds references to the same PeriodicEventHandler objects that were passed during construction.

4.6.4 Class javax.realtime.Scheduler

Declaration

@SCJAllowed
public abstract class Scheduler

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.

Static Methods

public static Schedulable getCurrentSo()

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

4.6.5 Class javax.realtime.PriorityScheduler

Declaration

@SCJAllowed
public class PriorityScheduler **extends** Scheduler

Description

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

Methods

@SCJAllowed
public int getMaxPriority()

Returns the maximum software real-time priority supported by this virtual machine.

@SCJAllowed
public int getNormPriority()

Returns the normal software real-time priority supported by this virtual machine.

@SCJAllowed
public int getMinPriority()

Returns the minimum software real-time priority supported by this virtual machine.

4.6.6 Class javax.safetycritical.PriorityScheduler

Declaration

@SCJAllowed
public class PriorityScheduler **extends** javax.realtime.PriorityScheduler

Description

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

Methods

@SCJAllowed
public int getMaxHardwarePriority()

Returns the maximum hardware real-time priority supported by this virtual machine.

@SCJAllowed
public int getMinHardwarePriority()

Returns the minimum hardware real-time priority supported by this virtual machine.

4.6.7 Class javax.realtime.AffinitySet

Declaration

@SCJAllowed
public final class AffinitySet

Description

This class is the API for all processor-affinity-related aspects of SCJ.

Static Methods

@SCJAllowed(LEVEL_2)
public static AffinitySet generate(BitSet bitSet)

Returns an AffinitySet representing a subset of the processors in the system. The returned object may be dynamically created in the current memory area or preallocated in immortal memory.

Throws NullPointerException if bitSet is null, and java.lang.IllegalArgumentException if bitSet is not a valid set of processors.

@SCJAllowed(LEVEL_1)
public static 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(LEVEL_2)
public static 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 NullPointerException if thread is null.

@SCJAllowed(LEVEL_1)
public static BitSet getAvailableProcessors()

Equivalent to getAvailableProcessors(BitSet dest) with a null argument.

@SCJAllowed(LEVEL_1)
public static BitSet getAvailableProcessors(BitSet dest)

Returns the set of processors available to the SCJ application either in dest, or if dest is null, the returned object may be dynamically created in the current memory area or preallocated in immortal memory.

@SCJAllowed(LEVEL_1)
public static AffinitySet getNoHeapSoDefaultAffinity()

Returns the default AffinitySet representing the set of processors on which no-heap schedulable objects can be scheduled. The returned object may be dynamically created in the current memory area or preallocated in immortal memory.

@SCJAllowed(LEVEL_1)
public static int getPredefinedAffinitySetCount()

Returns the size of the predefined affinity sets.

@SCJAllowed(LEVEL_1)
public static AffinitySet[] getPredefinedAffinitySets()

Equivalent to getPredefinedAffinitySets(AffinitySet[] dest) with a null argument.

@SCJAllowed(LEVEL_1)
public static AffinitySet[] getPredefinedAffinitySets(AffinitySet[] dest)

Returns an array of predefined AffinitySet, either in dest, or if dest is null, the returned object may be dynamically created in the current memory area or preallocated in immortal memory.

Throws java.lang.IllegalArgumentException if dest is not large enough to hold the set.

@SCJAllowed(LEVEL_1)
public static void setProcessorAffinity(AffinitySet set, BoundAsyncEventHandler handler)

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

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

@SCJAllowed(LEVEL_2)

public static void setProcessorAffinity(AffinitySet set, Thread thread)

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

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

Methods

@SCJAllowed(LEVEL_1)
public final BitSet getBitSet()

Equivalent to getProcessors(BitSet dest) with a null argument.

@SCJAllowed(LEVEL_1)
public final BitSet getProcessors(BitSet dest)

Returns the set of processors associated with this Affinity set, either in dest, or if dest is null, the returned object may be dynamically created in the current memory area or preallocated in immortal memory.

@SCJAllowed(LEVEL_2)
public final boolean isProcessorInSet(**int** processorNumber)

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

4.6.8 Class jaxax.safetycritical.Services

Declaration

@SCJAllowed
public class Services

Description

This class provides a collection of static helper methods.

Static Methods

@SCJAllowed(LEVEL_1)
public static int getDefaultCeiling();

Returns the default ceiling priority. The default ceiling priority is the PriorityScheduler.getMaxPriority.[3]

@SCJAllowed(LEVEL_1)
@SCJRestricted(INITIALIZATION)
public static void setCeiling(Object o, **int** pri);

Sets the ceiling priority of the first argument. The priority can be in the software or hardware priority range. Ceiling priorities are immutable.

[3] Although it is assumed that this can be changed with a virtual machine configuration option.

@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 dependent manner.

static void overwriteBackTraceAssociation(**Class** _class)

This method is invoked by infrastructure to change the association for the thread-local stack back trace buffer to the Class that represents a Throwable that has crossed its scope boundary, at the time that Throwable is replaced with a ThrowBoundaryError.

@SCJAllowed(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 javax.safetycritical.PriorityScheduler.getMaxPriority().

Throws IllegalArgumentException if unsupported InterruptId

@SCJAllowed
public static Level getDeploymentLevel()

Returns the deployment level.

@SCJAllowed(LEVEL_2)
@SCJMayBlock\issue{check}
public static void delay(HighResolutionTime delay)

This is like sleep except that it is not interruptible and it uses a HighResolutionTime.

Parameter if delay is a RelativeTime type then it represents the number of milliseconds and nanoseconds to suspend. If it is an AbsoluteTime type then delay is the absolute time at which the delay finishes. If delay is time in the past, the method returns immediately.

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

@SCJAllowed(LEVEL_2)
public static void delay(**int** delay)

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

Parameter delay is the the number of nanoseconds to suspend.

```
@ICS
@SCJAllowed(LEVEL_1)
public static void spin(HighResolutionTime delay)
```

Busy wait spinning loop.

Parameter if delay is a RelativeTime type then it represents the number of milliseconds and nanoseconds to spin. If it is an AbsoluteTime type then delay is the absolute time at which the spin finishes. If delay is time in the past, the method returns immediately.

```
@SCJAllowed(LEVEL_1)
public static void spin(int nanos) {}
```

Spin for nanos nanoseconds.

4.7 Rationale

Traditional safety critical systems have been small and sequential, relying on cyclic executive scheduling to manually interleave the execution of any activities with time constraints. Demonstration that timeliness requirements have been met has been through construction and testing. The limitations of this approach are well known.

As systems have become larger, there has been a gradual migration to computational models that support simple concurrent activities (be they, threads, tasks, event handlers etc) that share an address space with each other. Whereas testing may have 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 firm mathematical foundation, and are accepted by 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 for 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 including as evidence in any certification process for applications written according to this specification.

Specifying subsets of languages for use in safety system system is accepted practice, as to is constraining the way that subset is used. The Ada programming language 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 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, effectively is a static cyclic scheduler, where as levels 1 and 2 offers more dynamic flexible scheduling.

4.7.1 Scheduling and Synchronization Issues

For schedulability analysis, all non periodic activities must have bound minimum interarrival time. The SCJ specification does not provide the policing of inter-arrival times. In the RTSJ the use of sporadic release parameters requires that the implementation supports policing. Hence, the SCJ specification uses the aperiodic parameter class.

The priority ceiling emulation (PCE) protocol is optional in the RTSJ as many real-time OSs only support 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 execute lock retaining blocking operations) because it has an efficient implementation and has the potential to guarantee that the program is deadlock free. It also ensures that a schedulable object is blocked only once (at the start of its execution request).

SCJ only supports the priority ceiling emulation protocol. As the priority ceiling emulation protocol is optional in the RTSJ and compulsory in SCJ, SCJ defines it own interface. It simply provides a static method in the javax.safetycritical.Services class that just allows 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 blocking operations, for example the sleep and join methods when called from synchronized code. Furthermore, nested synchronized calls that call the wait method can release only one of the locks being held. For these reasons, self suspension of any type is not allowed at all Levels. At Level 2, if the use of wait method is allowed. The following approaches are possible.

1. Prohibit all nested synchronized calls. This seems draconian.
2. 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 with the standard Java semantics. On a single processor system, the PCE protocol would have to degrade to priority inheritance in this case (hence multiple possible blocking and the potential for deadlock). For multiprocessor systems, spinning for a lock would no longer be bounded.

4. Allow, but provide an annotation to indicate when synchronized code is suspension free.

4.7.2 Multiprocessors

Although the techniques for analyzing the timing properties of multiprocessor systems are still in their infancy, there is general acceptance on the growing importance of multicore platforms for real-time and embedded systems. For this reason, this specification provides support for programming multiprocessor platforms.

On a single processor, the priority ceiling emulation protocols has the following properties *if schedulable objects do not self suspend holding the lock*:

- no deadlocks can occur from the use of Java monitors.
- each schedulable object can be blocked at most once during its release as a result of sharing a Java monitor with another schedulable object.

The ceiling of each shared object is at least the maximum of all the schedulable objects that access that shared object.

On a multi-processor system, 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 whilst holding the monitor lock, then blocking can be bounded but the absence of deadlock cannot be assured by the protocol alone.

The usual approach to waiting for a lock that is help by a schedulable object on a different processors is to spin (busy-wait). There are different approaches that can be used by an implementation, for example, maintaining a FIFO/Priority queue of spinning processors, and whether the processors spin non-preemptively. SCJ does not mandate any particular approach but requires an implementation to document its approach.

To avoid unbounded priority inversion it is necessary to carefully set the ceiling values.

On a Level 1 system, the processors are fully partitioned using the scheduling allocation domain concept. The ceilings of every 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 an scheduling allocation domain, the priority of the ceiling 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 block by calling the wait method, results in the outer lock being held. Usually, in multiprocessor systems, this should 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 Multi-Processors

While feasibility analysis techniques are mature for single processor systems, they are less mature for multi-processor systems. Consequently, SCJ take a very conservative approach. SCJ introduces the notion of a *scheduling allocation domain*.

Scheduling allocation domains are represented by AffinitySets. As a schedulable object can only have one affinity set, it follows that each schedulable object can be executed only in a single allocation domain.

At Level 0, each scheduling allocation domain is implemented as a cyclic scheduler.

At Level 1, each scheduling allocation domain is scheduled using fixed priority preemptive scheduling. The feasibility analysis is equivalent to single processor feasibility analysis.

At level 2, 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.

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 from 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 release 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 second contribution to release jitter is also

related to the clock/timer. It is the duration of interval between T' being signaled by the clock/timer and the time this event is noticed by the underlying operating system or platform (perhaps because interrupts have been have been disabled). Figure 4.5 taken from [**?**] illustrates the delays that can occur.

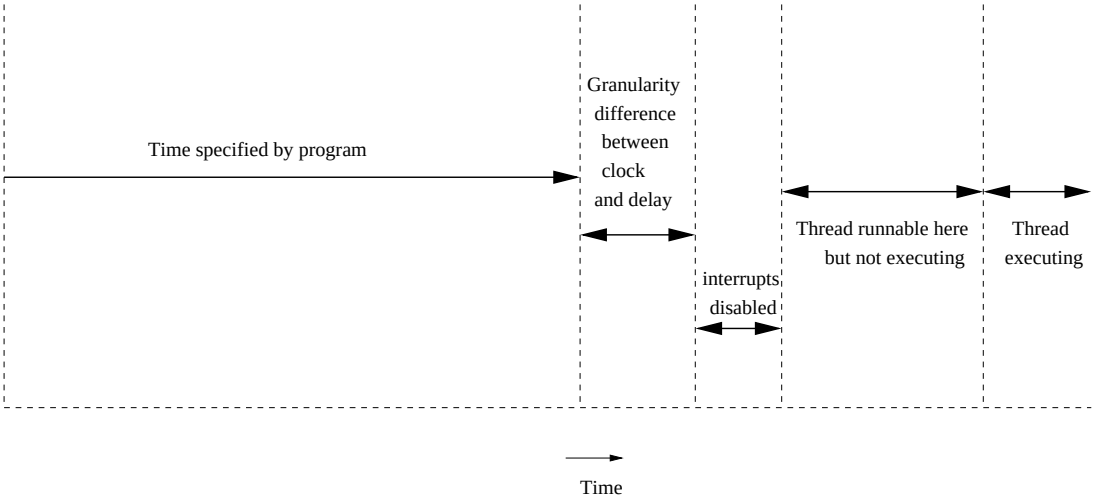

Figure 4.5: Granularity of delays (taken from [**?**])

A compliant implementation of SCJ should document the maximum value of Δ for the real-time clock.

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 facility is supported using a time-triggered event. As explained in Section 4.7.4, all time-triggered computation can suffer from release jitter. Hence, any deadline miss handler may not be released until sometime after the deadline has expired. The handlers actual execution will depend on its priority relative to other schedulable objects.

The second 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 the return of the handleAsyncEvent method. When this method returns, the infrastructure reschedules/cancels the timing 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 rescheduling/canceling of the timer event. Hence a deadline miss could be signalled when arguably the application had performed all of its computation.

For the above reasons, the SCJ deadline miss detection should be used with caution.

4.8 Compatibility

The following incompatibilities exist with RTSJ Version 1.1.

- PCE is the default monitor control policy in SCJ where as priority inheritance is the default in the RTSJ
- There are no facilities in the RTSJ to facilitate the specification of the SCJ StorageParameters.

Chapter 5

Interaction with External Devices

This chapter presents the facilities provided by SCJ to aid in the handling of interrupts and accessing memory-mapped (and port-mapped) input/output devices. The former is supported by the Happening class hierarchy and the latter by the raw memory access-related classes.

5.1 Happenings and Interrupt Handling

5.1.1 Semantics and Requirements

In the RTSJ and in SCJ, all external events are represented by *happenings*[1]. A happening represents a class of events that is detected by the hardware (interrupts), the real-time JVM or the system software.

Happenings may be assigned unique names and ids by the application, or the system will assign names and ids when they are not provided. The name space for system names is all strings beginning with the bullet character (\ u2022). The name space for system assigned ids is all integers less than zero.

In the RTSJ only second level handlers for happenings are supported and can be mapped to the release of asynchronous event handlers. In SCJ, both first-level and second-level handlers are supported. First-level handlers are associated with interrupt happenings (an extension to the RTSJ framework). Furthermore, all handlers are managed in the context of a mission.

Unlike the RTSJ, SCJ fully defines its underling model of happenings. The model is heavily influenced by the Ada interrupt handling model, and borrows most of its definitions from that model. The following definitions are used in the SCJ model.

[1]The happening model in jsr282 may change in the near future. If it does there many be some repercussions for the material presented in this chapter.

- An *occurrence* of an external event consists of its *generation* and *delivery*.
- Generation of the external event is the mechanism in the underlying hardware, real-time JVM or system that makes the external event available to the Java program.
- Delivery is the action that invokes the associated trigger method in response to the occurrence of the external event. This may be performed by the JVM or application native code linked with the JVM.
- Between generation and delivery, the external event is *pending*.
- Some or all external event occurrences may be inhibited. When an external event occurrence is inhibited, all occurrences of that event are prevented from being delivered.
- Certain implementation-defined external events are *reserved*. Reserved external events are either external events for which user-defined Happenings are not supported, or those that already have registered Happenings by some other implementation-defined means. A clock interrupt is an example of reserved external event.
- Happenings can be connected to non-reserved external events. While connected, the happening is said to be *registered* to that happening. The trigger method, is invoked upon delivery of a registered happening occurrence.
- While a happening is registered to an external event, it's trigger method is called *once* for each delivery of that external event. While the trigger method executes, the corresponding external event is inhibited. While an external event is inhibited, all occurrences of that external event are prevented from being delivered. Whether such occurrences remain pending or are lost is implementation defined.
- Each external event has a default implementation-defined registered Happening.
- An exception propagated from a trigger method that is invoked by an external handler results in the uncaughtException method being called in the associated ManageInterruptHandler class.
-

The implementation shall document the following items:

1. For each external event, whether it can be inhibited or not, and the effects of attaching Happenings to non inhibitable external events (if this is permitted)

2. Which run-time stack the trigger method uses when it executes; if this is configurable, what is the mechanism to do so; how to specify how much space to reserve on that stack.

3. Any implementation- or hardware-specific activity that happens before a user-defined happenings gets control (e.g., reading device registers, acknowledging

devices).

4. The state (inhibited/uninhibited) of the non-reserved external events when the program starts; if some external events are uninhibited, what is the mechanism a program can use to protect itself before it can register the corresponding Happening.

5. The treatment of external event occurrences that are generated while the external event 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 external event (for example, a hardware trap resulting from a segmentation error), and the mapping between the external event and the predefined exceptions.

7. On a multi-processor, the rules governing the delivery of an external event occurrence to a particular processor.

Writing first-level interrupt handlers requires a restricted use of the Java language. To support their introduction, SCJ defines the notion of an *Interrupt Safe*(IS) method. An IS method is one that does no allocation and does not block (see Chapter 9). Furthermore, any native code called from an IS method should not block or interact with the underlying operating system (if present). The following are examples of methods that are IS.

- Happening.Trigger
- Object.notify/notifyAll
- RawMemory access

Note that any exception thrown will be lost if the IS method is called from a first-level interrupt handler.

SCJ also defines the notion of interrupt priorities. Interrupt priorities can only be used to define ceiling priorities.

The revised happening framework is shown in Figure 5.1.

The Use Case diagram in 5.2 shows how the classes are used during mission initialization and interrupt handling processing.

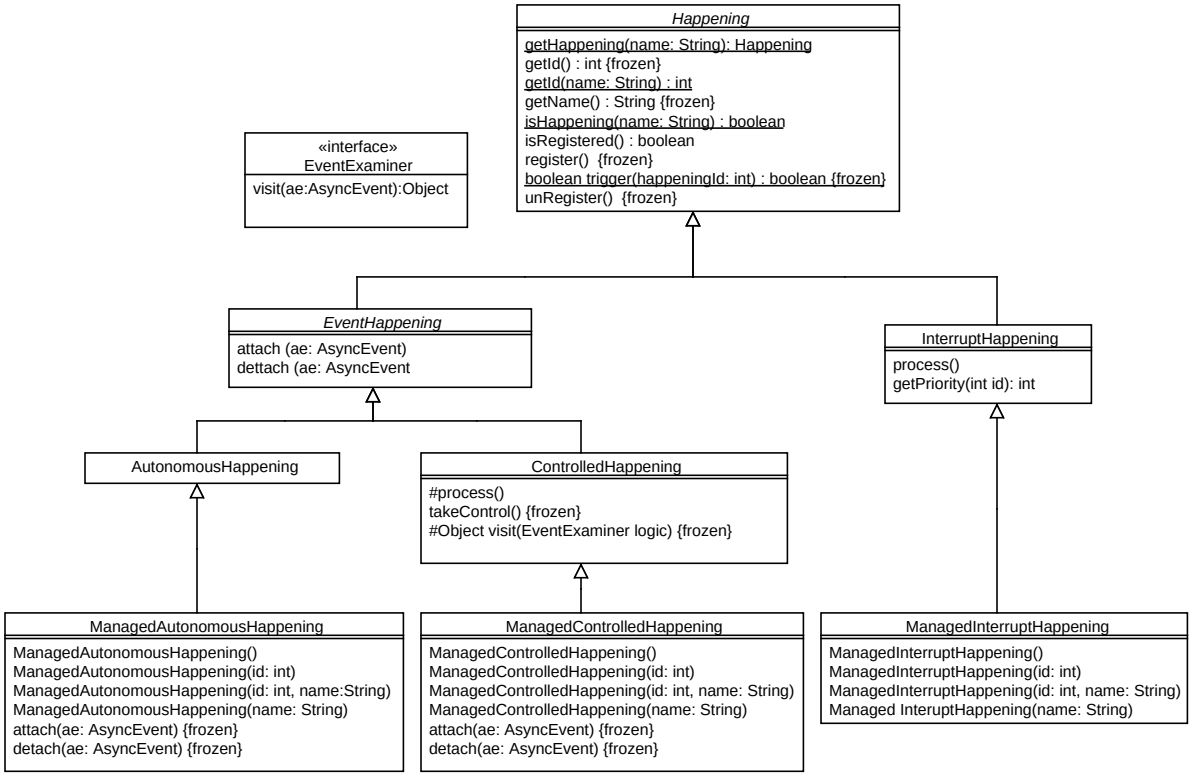

Figure 5.1: Happening classes

5.1.2 Level Considerations

Level 0

- Non-reserved happenings of any kind are prohibited at Level 0. All interaction with the external embedded environment must be performed in a synchronous manner.

Level 1

- ManagedInterruptHappenings and ManagedAutonomousHappenings are supported.
- Only one asynchronous event can be attached to each autonomous happening.
- Each autonomous happening may be attached to only one asynchronous event.
- The registration of a happening can only be performed during the mission initialization phase by the mission manager. The deregistration of an asynchronous event can only be performed during the cleanup phase.

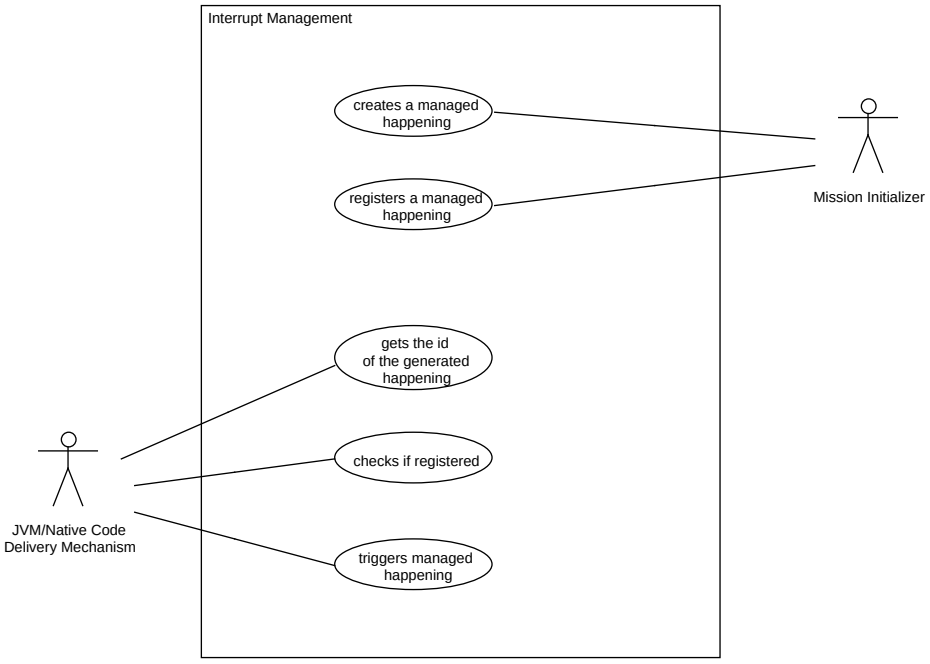

Figure 5.2: Managed Happening Use Case

- The attachment of a happening to an asynchronous event can only be performed during the mission initialization phase. The detachment of an asynchronous event can only be performed during the cleanup phase.
- The RTSJ bindTo mechanisms is not supported. item

Level 2

- ManagedControlledHappenings are supported.
- More than one asynchronous event can be attached to each autonomous happening.

5.2 The Happening Class Hierarchy

5.2.1 Class javax.realtime.Happening

Declaration

```
@SCJAllowed(LEVEL_1)
public abstract class Happening {
```

Description

In SCJ, all external events are represented by the Happening class.

Static Methods

@SCJAllowed(LEVEL_1)
public static Happening getHappening(String name)

Returns the happening denoted by name.

@SCJAllowed(LEVEL_1)
public static int getId(java.lang.String name);

Returns the ID of the happening denoted by name. If there is no happening with that name returns 0.

@SCJAllowed(LEVEL_1)
public static boolean isHappening(java.lang.String name)

Returns true if there a Happening denoted by name. False otherwize.

@SCJAllowed(LEVEL_1)
public static final boolean trigger(**int** happeningId) {**return** true; }

Causes the happening corresponding to happeningId to occur. The trigger method is responsible for determining the type of happening. For autonomous happenings, the associated asynchronous events are fired. For controlled happenings or interrupt happenings, the process method is called. *Returns* true if a happening with id happeningId was found, false otherwise. **Methods**

@SCJAllowed(LEVEL_1)
public final int getId()

Returns the id of this happening.

@SCJAllowed(LEVEL_1)
public final String getName()

Returns the string name of this happening.

@SCJAllowed(LEVEL_1)
public boolean isRegistered()

Returns true if this happening is presently registered. False otherwize.

@SCJAllowed(LEVEL_1)
@SCJRestricted(INITIALIZATION)
public final void register();

Register this Happening.

Throws **IllegalStateException** if called from outside the mission initialization phase.

@SCJAllowed(LEVEL_1)

```
@SCJRestricted(CLEANUP)
public final void unRegister();
```

Unregister this Happening.

Throws **IllegalStateException** if called from outside the mission clean-up phase.

5.2.2 javax.realtime.EventHappening

Declaration

```
@SCJAllowed(LEVEL_1)
public abstract class EventHappening extends Happening
```

Description

Event happenings are happenings that can be associated with asynchronous events (via the attach method).

Constructors

Methods

```
@SCJAllowed(LEVEL_1)
@SCJRestricted(INITIALIZATION)
public void attach(AsyncEvent ae)
```

Attach the AsyncEvent ae to this Happening.

```
@SCJAllowed(LEVEL_1)
@SCJRestricted(CLEANUP)
public void detach(AsyncEvent ae)
```

Detach the AsyncEvent ae from this Happening.

Throws **IllegalStateException** if called from outside the mission initialization phase.

5.2.3 javax.realtime.AutonomousHappening

Declaration

```
@SCJAllowed(LEVEL_1)
public class AutonomousHappening extends EventHappening
```

Description Autonomous happenings are those that when triggered automatically fire the attached asynchronous events. In SCJ, all happenings are managed, hence there are no visible constructors for the class.

5.2.4 javax.safetycritical.ManagedAutonomousHappening

Declaration

```
@SCJAllowed(LEVEL_1)
public class ManagedAutonomousHappening extends AutonomousHappening
```

Description Managed autonomous happenings automatically, on call of a constructor, associates itself with the current mission.

Constructors

```
@SCJAllowed(LEVEL_1)
public ManagedAutonomousHappening()
```

Creates a happening in the current memory area with a system assigned name and id.

```
@SCJAllowed(LEVEL_1)
public ManagedAutonomousHappening(int id) {};
```

Creates a Happening in the current memory area with the specified id and a system-assigned name.

```
@SCJAllowed(LEVEL_1)
public ManagedAutonomousHappening(int id, String name) {};
```

Creates a Happening in the current memory area with the name and id given.

```
@SCJAllowed(LEVEL_1)
public ManagedAutonomousHappening(String name) {};
```

Creates a Happening in the current memory area with the given name and a system-assigned id.

Methods

```
@SCJAllowed(LEVEL_1)
@SCJRestricted(INITIALIZATION)
public final void attach(AsyncEvent ae)
```

Attach the AsyncEvent ae to this Happening.

Throws **IllegalStateException** if called from outside the mission initialization phase.

```
@SCJAllowed(LEVEL_1)
@SCJRestricted(CLEANUP)
public final void detach(AsyncEvent ae)
```

Detach the AsyncEvent ae from this Happening.

Throws **IllegalStateException** if called from outside the mission clean-up phase.

5.2.5 javax.realtime.EventExaminer

Declaration

```
@SCJAllowed(LEVEL_2)
public interface EventExaminer
```

Description

An interface used in conjunction with controlled happenings

Methods

```
@SCJAllowed(LEVEL_2)
Object visit(AsyncEvent ae);
```

5.2.6 javax.realtime.ControlledHappening

Declaration

```
@SCJAllowed(LEVEL_2)
public class ControlledHappening extends EventHappening
```

Description Controlled happenings are those that when triggered allow the application to take control. The application schedulable object calls one of the takeControl methods. That method calls the process method each time the happening is triggered. The takeControl method returns when the happening is unregistered.

In SCJ, all happenings are managed, hence there are no visible constructors for the class. **Methods**

@SCJAllowed(LEVEL_2) **protected void** process()

This method should be overridden if the application wishes to use some non-default behavior on being triggered, for example by adding some operation before or after invoking super.process, by ignoring attached events, or by using visit(EventExaminer) to fire a subset of the attached async events or take some other action for the attached async events.

The default process method behaves effectively like: visit(logic) where the EventExaminer.visit(AsyncEvent) method in logic is

```
ae.fire();
return null;
```

```
@SCJAllowed(LEVEL_2)
public final void takeControl()
```

The application supplies a SO to the happening using this method. The happening shall use the calling SO to process happenings. The takeControl method does not return until the happening is deregistered. The takeControl method behaves effectively as if it were implemented:

```
while(isRegistered){
    // waitForTrigger
    process()
}
```

Throws java.lang.IllegalStateException if this happening is already controlled by an SO, if the calling SO's current memory area is not the memory area containing this happening, or if the happening is not registered.

```
@SCJAllowed(LEVEL_2)
protected final Object visit(EventExaminer logic)
```

Executes logic on each attached async event until either logic.visit returns non-null or all happenings have been visited.

Returns null if all async events were visited, the non-null object reference returned by the last call to the async event examiner otherwise.

Throws java.lang.IllegalStateException - if the caller is not the SO controlling this happening, see the takeControl method.

5.2.7 javax.safetycritical.ManagedControlledHappening

Declaration

```
@SCJAllowed(LEVEL_2)
public class ManagedControlledHappening extends ControlledHappening
```

Description Managed controlled happenings automatically, on call of a constructor, associates itself with the current mission.

Constructors

```
@SCJAllowed(LEVEL_2)
public ManagedControlledHappening()
```

Creates a happening in the current memory area with a system assigned name and id.

```
@SCJAllowed(LEVEL_2)
public ManagedControlledHappening(int id)
```

Creates a Happening in the current memory area with the specified id and a system-assigned name.

```
@SCJAllowed(LEVEL_2)
public ManagedControlledHappening(int id, String name)
```

Creates a Happening in the current memory area with the name and id given.

@SCJAllowed(LEVEL_2)
public ManagedControlledHappening(String name)

Creates a Happening in the current memory area with the specified id and a system-assigned name.

@SCJAllowed(LEVEL_2)
@SCJRestricted(INITIALIZATION)
public final void attach(AsyncEvent ae)

Attach the AsyncEvent ae to this Happening.

Throws **IllegalStateException** if called from outside the mission initialization phase.

@SCJAllowed(LEVEL_2)
@SCJRestricted(CLEANUP)
public final void detach(AsyncEvent ae)

Detach the AsyncEvent ae from this Happening.

Throws **IllegalStateException** if called from outside the mission clean-up phase.

5.2.8 javax.realtime.InterruptHappening

Declaration

@SCJAllowed(LEVEL_1)
public class InterruptHappening **extends** Happening

In SCJ, all interrupts are managed, hence there are no visible constructors for the class.

Methods

@SCJAllowed(LEVEL_1)
protected void process()

This method should be overridden to provide the interrupt handler.

@SCJAllowed(LEVEL_1)
public final int getPriority()

Returns the priority at which the process method is executed.

5.2.9 javax.safetycritical.ManagedInterruptHappening

Declaration

```
@SCJAllowed(LEVEL_1)
public class ManagedInterruptHappening extends InterruptHappening
```

In SCJ all interrupt handlers must be known by the mission manager, hence applications use classes that are based on javax.safetycritical.ManagedInterruptHappening.

Constructors

```
@SCJAllowed(LEVEL_1)
public ManagedInterruptHappening()
```

Creates a Happening in the current memory area with a system assigned name and id.

```
@SCJAllowed(LEVEL_1)
public ManagedInterruptHappening(int id) {};
```

Creates a Happening in the current memory area with the specified id and a system-assigned name.

```
@SCJAllowed(LEVEL_1)
public ManagedInterruptHappening(int id, String name) {};
```

Creates a Happening in the current memory area with the given name and id given.

```
@SCJAllowed(LEVEL_1)
public ManagedInterruptHappening(String name) {};
```

Creates a Happening in the current memory area with the given name and a system-assigned id.

Methods

```
@SCJAllowed(LEVEL_1)
public void uncaughtException(Exception E)
```

Called by the Infrastructure if an interrupt handler throws an uncaught exception

5.3 Raw Memory Access

5.3.1 Semantics and Requirements

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 means of 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.

SCJ restricts the RTSJ API by not requiring any of the classes related to *physical memory* and *removable memory*.

- 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. It is used to access input and output device registers when such registers are memory mapped.
- The raw memory name IO_PORT_MAPPED facilitates access to location 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 location that are used to access input and output device registers when such registers are memory mapped.
- The raw memory name DMA_ACCESS facilitates access to memory location 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 policed by implementation-defined objects that are created by implementation-defined factory objects. Each factory is identified by its raw memory name.
- Only Java integral types are supported.

An overview of the supported classes and interfaces is shown in Figure: 5.3 and their interactions in Figure: 5.4.

5.3.2 Level Considerations

The defined facilities are available at all levels.

5.3.3 javax.realtime.RawMemoryName

Declaration

```
@SCJAllowed(LEVEL_0)
public interface RawMemoryName
```

Description RawMemoryName is a tagging interface for objects that identify raw memory types.

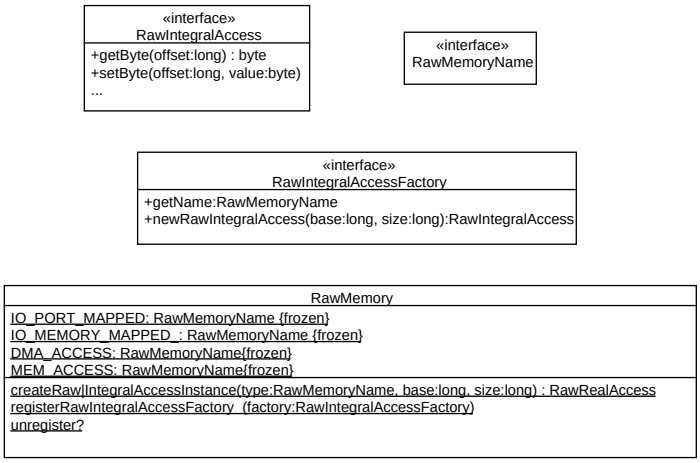

Figure 5.3: Raw memory classes

5.3.4 javax.realtime.RawIntegralAccess

Declaration

@SCJAllowed(LEVEL_0)
public interface RawIntegralAccess

Description

An interface that facilitates access to raw memory using with scalar data types. Implementation-defined objects that implement this interface police access to a contiguous area of physical memory. Offsets are used to access instance or arrays of the primitive Java scalar types.

SCJ supports this interface in its entirety. The full interface is given in Section **??**, here an abbreviated version is shown:

Methods

@SCJAllowed(LEVEL_0)
@SCJRestricted{InterruptSafe}
public byte getByte(**long** offset);

Get the byte at the given offset in the raw memory area associated with this object..

Parameter offset is the offset at which to read the byte.

Returns the byte from raw memory.

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

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

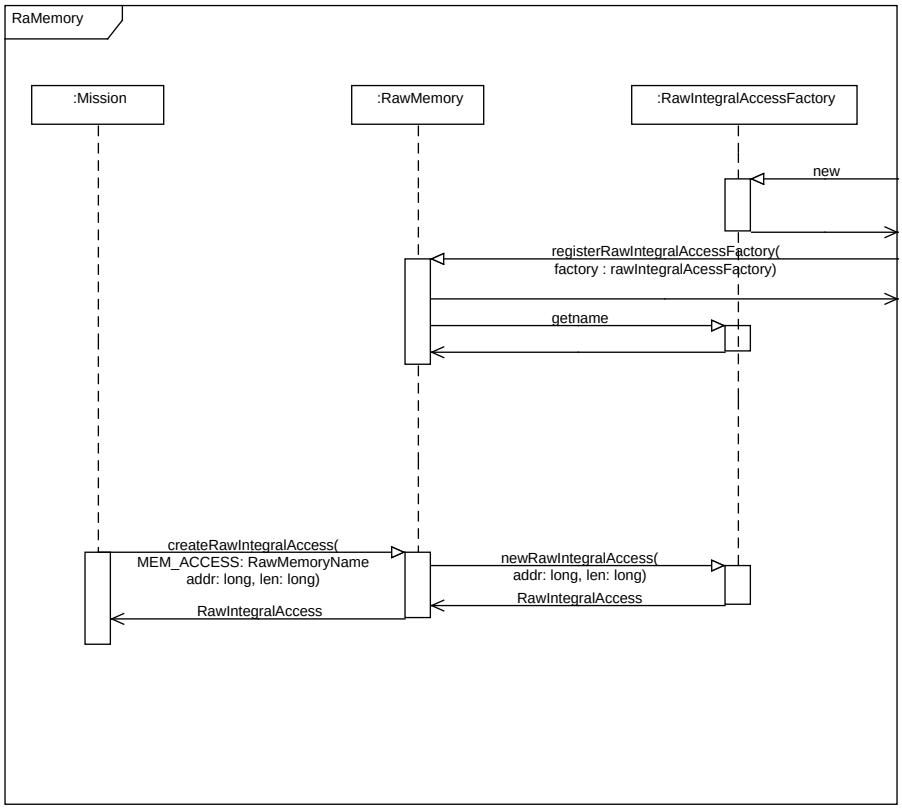

Figure 5.4: Raw memory classes interactions

@SCJAllowed(LEVEL_0)
@SCJRestricted{InterruptSafe}
public void getBytes(**long** offset, **byte**[] bytes, **int** low, **int** number);

Gets number bytes starting at the given offset and assigns them to the byte array passed starting at position low. Each byte is loaded from memory in a single atomic operation. Groups of bytes may be loaded together, but this is unspecified.

Caching of the memory access is controlled by the factory that created this RawIntegralAccess instance. If the memory is not cached, this method guarantees serialized access (that is, the memory access at the memory occurs in the same order as in the program. Multiple writes to the same location may not be coalesced.)

Parameter offset is the offset in bytes from the beginning of the raw memory from which to start loading.

Parameter bytes is the array into which the loaded items are placed.

Parameter low is the offset which is the starting point in the given array for the loaded items to be placed.

Parameter number is the number of items to load.

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

Throws SizeOutOfBoundsException if the bytes falls in an invalid address range. This is checked at every entry in the array to allow for the possibility that the memory area could be unmapped or remapped. The bytes array could, therefore, be partially updated if the raw memory is unmapped or remapped mid-method.

Throws java.lang.ArrayIndexOutOfBoundsException if low is less than 0 or greater than bytes.length - 1, or if low + number is greater than or equal to bytes.length.

```
@SCJAllowed(LEVEL_0)
@SCJRestricted{InterruptSafe}
public void setByte(long offset, byte value);
```

Sets the byte at the given offset in the raw memory area associated with this object.

This memory access may involve a load and a store, and it may have unspecified effects on surrounding bytes in the presence of concurrent access.

Caching of the memory access is controlled by the factory that created this RawIntegralAccess instance. If the memory is not cached, this method guarantees serialized access (that is, the memory access at the memory occurs in the same order as in the program. Multiple writes to the same location may not be coalesced.)

Parameter offset is the offset in bytes from the beginning of the raw memory area to which to write the byte.

Parameter value is the byte to write.

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

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

```
@SCJAllowed(LEVEL_0)
@SCJRestricted{InterruptSafe}
public void setBytes(long offset, byte[] bytes, int low, int number);
```

Sets number bytes starting at the given offset in the raw memory area associated with this object from the byte array passed starting at position low. This memory access may involve multiple load and a store operations, and it may have unspecified effects on surrounding bytes (even bytes in the range being stored) in the presence of concurrent access.

Caching of the memory access is controlled by the factory that created this RawIntegralAccess instance. If the memory is not cached, this method guarantees serialized access (that is, the memory access at the memory occurs in the same order as in the program. Multiple writes to the same location may not be coalesced.)

Parameter offset is the offset in bytes from the beginning of the raw memory area to which to start writing. *Parameter* bytes is the array from which the items are obtained. *Parameter* low is the offset which is the starting point in the given array for the items to be obtained. *Parameter* number is the number of items to write.

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

Throws SizeOutOfBoundsException if the object is not mapped, or if the a byte falls in an invalid address range.

Throws java.lang.ArrayIndexOutOfBoundsException if low is less than 0 or greater than bytes.length - 1, or if low + number is greater than or equal to bytes.length.

5.3.5 javax.realtime.RawIntegralAccessFactory

Declaration

@SCJAllowed(LEVEL_0)
public interface RawIntegralAccessFactory

Description An interface that describes factory classes that create classes that implement RawIntegralAccess.

Methods

@SCJAllowed(LEVEL_0)
public 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(LEVEL_0)
public RawIntegralAccess newIntegralAccess(**long** base, **long** size); ∗∗∗CHECK **THIS**

Gets an instance of a class that implements the RawIntegralAccess interface and can access raw memory starting at the base address and extending for size bytes. It need not be a new instance.

Returns an instance of RawIntegralAccess supporting access to the requested range of memory (and only that range of memory.)

Throws java.lang.IllegalArgumentException if base is negative, or size is not greater than zero.

Throws OffsetOutOfBoundsException if base is invalid.

*Throws*SizeOutOfBoundsException if size extends into an invalid range of memory.

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

Throws java.lang.OutOfMemoryError if any part of the memory starting at base for length size has already been allocated or claimed for raw access.

5.3.6 javax.realtime.RawMemory

Declaration

@SCJAllowed(LEVEL_0)
public final class RawMemory

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 registerAccessFactory and createRawIntegralAcessInstance methods. In SCJ, four raw-integral-access factories are supported: two for accessing the DEVICE 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). These can be accessed via static methods in the RawMemoryAccess class.

Static Fields

@SCJAllowed(LEVEL_0)
public static final RawMemoryName DMA_ACCESS

This raw memory name is used to call for access memory using DMA.

@SCJAllowed(LEVEL_0)
public static final RawMemoryName MEM_ACCESS

This raw memory name is used to call for access memory.

@SCJAllowed(LEVEL_0)
public static final RawMemoryName IO_PORT_MAPPED

This raw memory name is used to call for access to all I/O devices that are accessed by special instructions.

@SCJAllowed(LEVEL_0)
public static final RawMemoryName IO_MEM_MAPPED

This raw memory name is used to call for access to devices that are memory mapped.

Static Methods

@SCJAllowed(LEVEL_0)
public static RawIntegralAccess createRawIntegralInstance(
RawMemoryName type, **long** base, **long** size)

Create (or find) an immortal instance of a class that implements RawIntegralAccess and accesses (only) memory of type type in the address range described by base and size.

Returns an object that implements RawIntegralAccess and supports access to the specified range of physical memory.

Throws

- java.lang.IllegalArgumentException if base is negative, or size is not greater than zero.
- java.lang.SecurityException if application doesn't have permissions to access physical memory, the specified range of addresses, or the type of memory supported by this factory.
- OffsetOutOfBoundsException if base is invalid.
- SizeOutOfBoundsException if size extends into an invalid range of memory.
- MemoryTypeConflictException if base does not point to memory that matches the type served by this factory.
- java.lang.OutOfMemoryError if any part of the memory starting at base for length size has already been allocated or claimed for raw access.
- java.lang.InstantiationException
- java.lang.IllegalAccessException
- java.lang.reflect.InvocationTargetException

public static void registerAccessFactory(RawIntegralAccessFactory factory)

Make factory known to the factory as the class that should be used to access type RawIntegralAccessFactory.getName(). Only at most one factory per type is permitted (though one factory may handle many types). An attempt to add another factory will throw an illegal argument exception.

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

5.4 Rationale

Many safety critical real-time systems have to 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. Low-level device access is the topic of this chapter.

A device can be considered to be a processor performing a fixed task. A computer system can, therefore, 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. **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

It depends if RTSJ does support Interrupt happenings or not.

Chapter 6

Input and Output Model

Safety-critical systems often have limited input and output capabilities. The capabilities they do have are tailored to the task of the system. 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. Fortunately, Java Micro Edition provides a basis for a flexible I/O mechanism.

6.1 Semantics and Requirements

Since there is no I/O facility that can be found on every safety-critical system, what is needed is a flexible mechanism for adding I/O capabilities. The Micro Edition I/O Connector and Connection classes, with the StreamConnection, InputConnection, and OutputConnection interfaces, provide a good basis.

A Connection relates 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 a factory to create a connection of the proper type.

The protocol console defines the default console. The console can be used to read from and send output to some system-defined data source or sink, so that test harnesses can use them for getting feedback on whether tests succeed or fail.

6.2 Level Considerations

There are no level considerations for I/O.

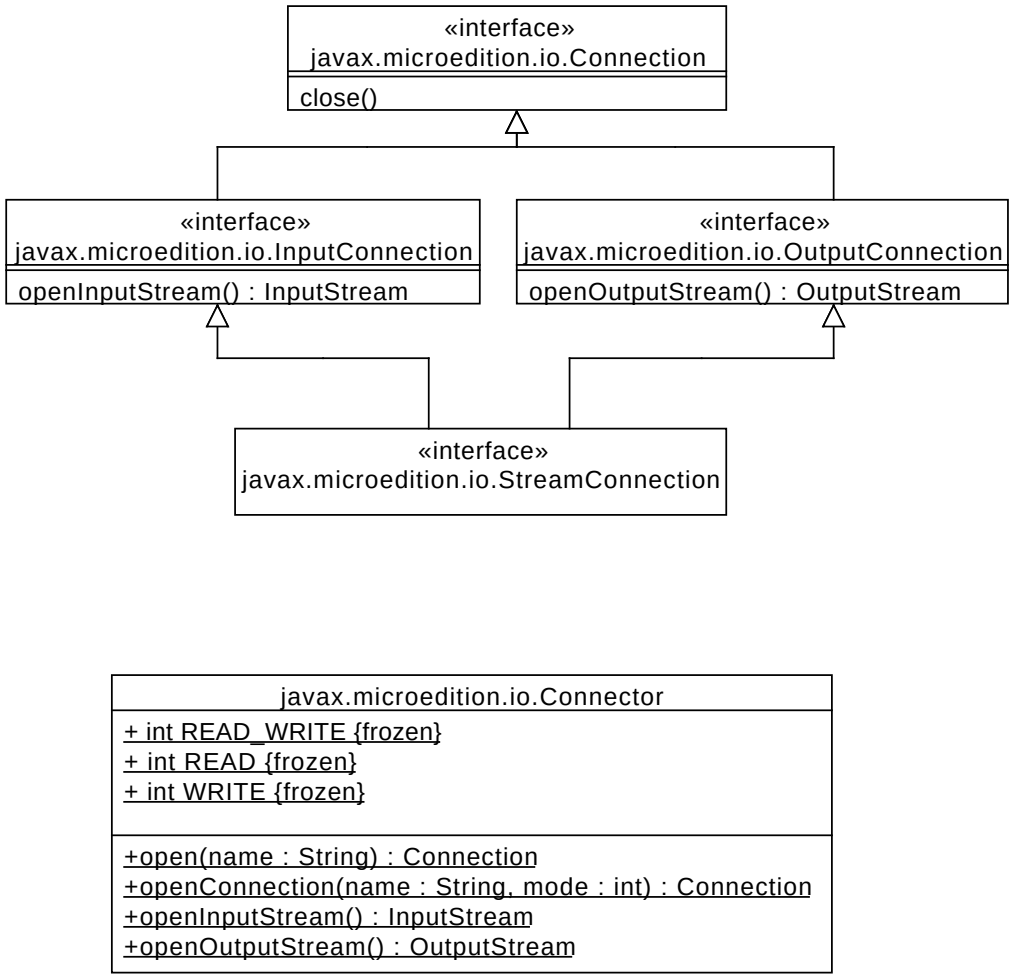

Figure 6.1: Interfaces and classes supporting streaming I/O

6.3 APIs

The API is a subset of the API as defined in package javax.microedition.io of the Java Micro Edition.

6.3.1 Interface javax.microedition.io.Connection

Declaration

```
@SCJAllowed
public abstract interface Connection
```

Description

A generic connection that just provides the ability to be closed. No open method is defined as the connection is opened by Connector.open().

Methods

```
@SCJAllowed
public void close() throws IOException
```

Close the connection. Closing an already closed connection has no effect.

6.3.2 Class javax.microedition.io.Connector

Declaration

```
@SCJAllowed
public class Connector
```

Description

This class is the factory for creating connection objects. The connection type and channel is defined by the parameter string, which describes the target in the URL form. The general form is scheme:[target][params] where scheme is the protocol name such as http. target specifies the destination address (e.g., a serial port number); and params can be used for additional parameters (e.g., the baud rate).

An SCJ implementation shall provide at least a console connection for debugging and test output. The parameter string for the console connection is "console:".

Constructors

No public visible constructors, as the class has only static factory methods.

Fields

```
@SCJAllowed
public final static int READ = 1;

@SCJAllowed
public final static int WRITE = 2;

@SCJAllowed
public final static int READ_WRITE = (READ|WRITE);
```

Access mode for read, write, and read/write passed to open().

Methods

```
@SCJAllowed
public static Connection open(String name)
  throws IllegalArgumentException, ConnectionNotFoundException, IOException
```

Create and open a Connection.

```
@SCJAllowed
public static Connection open(String name, int mode)
  throws IllegalArgumentException, ConnectionNotFoundException, IOException
```

Create and open a Connection.

```
@SCJAllowed
public static InputStream openInputStream(String name)
  throws IllegalArgumentException, ConnectionNotFoundException, IOException
```

Create and open a connection input stream.

```
@SCJAllowed
public static OutputStream openOutputStream(String name)
  throws IllegalArgumentException, ConnectionNotFoundException, IOException
```

Create and open a connection output stream.

6.3.3 Class javax.microedition.io.ConnectionNotFoundException

Declaration

```
@SCJAllowed
public class ConnectionNotFoundException extends Exception
```

Description

An exception to throw when the connection for a given URL cannot be created because the resources are not available or no factory exists.

Constructors

```
@SCJAllowed
public ConnectionNotFoundException(String message)
```

Create this exception with a text description.

```
@SCJAllowed
public ConnectionNotFoundException()
```

Create this exception with no description.

6.3.4 Interface javax.microedition.io.InputConnection

Declaration

```
@SCJAllowed
public interface InputConnection extends Connection
```

Description

An interface for connections that can input data.

Methods

```
@SCJAllowed
public InputStream openInputStream() throws IOException
```

Open and return an input stream for a connection.

6.3.5 Interface javax.microedition.io.OutputConnection

Declaration

```
@SCJAllowed
public interface OutputConnection extends Connection
```

An interface for connections that can output data.

Methods

```
@SCJAllowed
public OutputStream openOutputStream() throws IOException;
```

Open and return an output stream for a connection.

6.3.6 Interface javax.microedition.io.StreamConnection

Declaration

```
@SCJAllowed
public interface StreamConnection extends InputConnection, OutputConnection
```

An interface for Connections that can both read and write data.

6.4 Rationale

This API provides the best tradeoff between compatibility and light weight. Using the I/O facilities in java.io, java.net, and java.file, or java.nio would require too many classes, many of which could not be supported on all systems.

6.5 Compatibility

These classes are a subset of the Java Micro Edition connection framework.

Chapter 7

Memory Management

As with standard Java, all memory allocation in RTSJ and SCJ programs is performed from within an *allocation context*. This defines the space from which a new object should be taken. Whereas standard Java implicitly defines a single heap as allocation context, the RTSJ generalizes allocation contexts through the classe MemoryArea and the interface AllocationContext, where the heap is just one instance of this class and this interface respectively. Furthermore, additional allocation contexts are defined. SCJ restricts the use of allocation contexts defined by the RTSJ to the use of special subclasses of LTMemory: MissionMemory and PrivateMemory. In SCJ, neither of these areas can be created directly by calling its constructor.

7.1 Semantics and Requirements

As discussed in Chapter 3, SCJ supports the notion of a mission and a mission life cycle. An application with this model has three phases as shown in Figure 3.1: initialization, execution, and cleanup.

The data structures that are needed for a given mission are allocated in a special scoped memory area called *mission memory*. This scope remains active for the duration of the mission and acts like an immortal memory for that mission. Normally mission memory allocation takes place in the initialization phase and those structures persist during the life of the mission. Ephemeral structures are usually allocated in a *private scoped memory* during the execution phase.

Nested missions are supported, so an application may have more than one active mission memory.

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 are supported. Variable time scoped memory and the heap areas are not supported.
- A linear-time scope memory area (using the MissionMemory class) is provided which is entered at the beginning of mission initialization and exited after mission clean-up before the next mission is initialized.
- Objects allocated in the initialization phase are never collected throughout the duration of a given mission unless they are explicitly allocated in a private linear-time scoped memory area (using the PrivateMemory class).
- A private memory area is owned by a single schedulable object and it can only be entered by that schedulable object.
- Every schedulable object has one preallocated private memory area and all allocations performed during a release (see Chapter 4) of that schedulable object will, by default, be performed in this private scoped memory. The memory allocated to objects created in this private scoped memory will be reclaimed at the end of the release.
- Schedulable objects can create and enter into nested private memory areas. These memory areas are, by definition, not shared and must be entered directly from the scope in which there are created.
- The backing store for a scoped is taken from the backing store reservation of it's schedulable object.
- Backing store is managed via reservations of backing store for use by schedulable objects, where the initial schedulable object partitions portions of its backing store reservation to pass on to schedulable objects created in its thread of control.
- Backing Store size reservation is managed via StorageParameters objects.
- SCJ does not support object finalizers. The same effect can be obtained with try statement that includes a finally clause for any given scope.
- SCJ conforms to the Java memory model. In addition, access to raw memory is considered as volatile access (see Section 5.3).

Figure 7.1 illustrates the limited use of hierarchical memory areas in the RTSJ.

7.2 Level Considerations

7.2.1 Level 0

Level 0 supports only a single mission sequence. The same mission memory is reused for each mission in the sequence, however, the size of the mission mem-

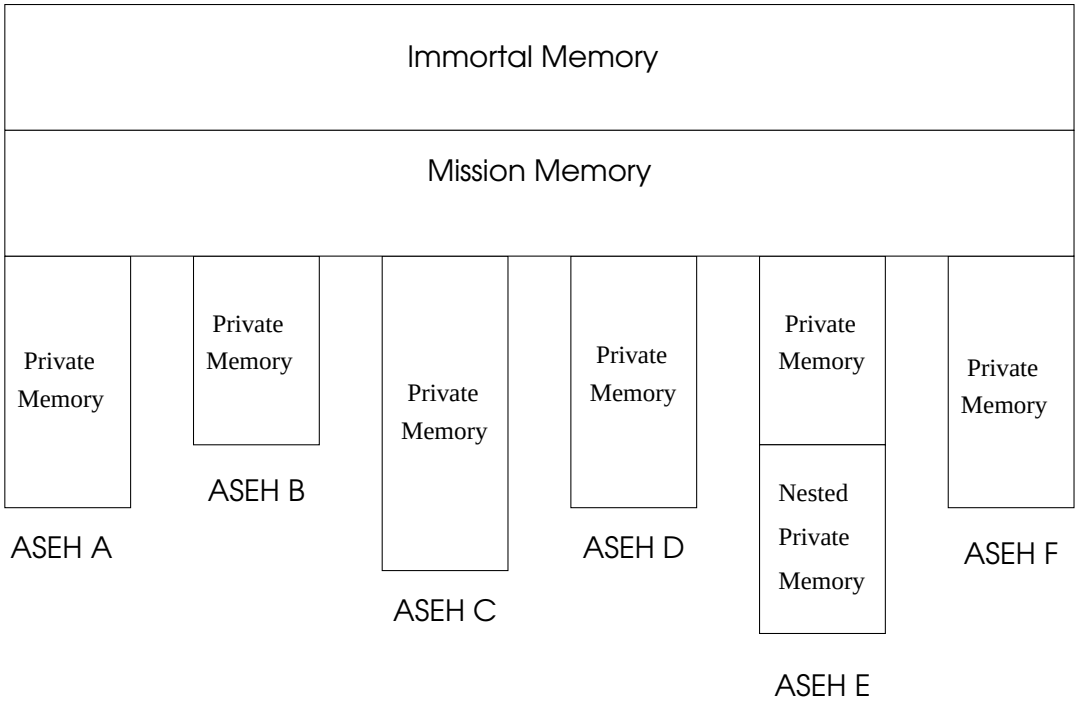

Figure 7.1: Example of Memory Areas used by a Level 1 Application

ory may change between missions. The contents is cleared between missions. Each AsyncEventHandler has its own PrivateMemory that is entered into for the duration of its handleAsyncEvent method called within its frame. This corresponds to release and completion in higher levels. The application programmer may allocate PrivateMemory within a frame, so simple nesting of scopes is possible.

7.2.2 Level 1

Level 1 supports sequences of missions and private memory for each handler as well, but the handlers are run asynchronously.

7.2.3 Level 2

Level 2 adds the ability to nest MissionMemory. A MissionMemory may not be created or entered from a PrivateMemory. A nested MissionMemory is created by its MissionSequencer.

7.3 Memory related APIs

SCJ supports only a subset of the RTSJ memory model. Consequently many of the methods are absent (and, therefore complexity of the overall model is reduced). The application can only create SCJ-defined private memory areas. Figure 7.2 provides an overview of the supported interfaces and classes.

7.3.1 Interface javax.realtime.AllocationContext

Declaration

@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.

Methods

@SCJAllowed
public void enter(Runnable logic)

Execute logic with this memory area as the current allocation context.

@SCJAllowed
public void executeInArea(Runnable logic)

Execute logic with this memory area as the current allocation context.

@SCJAllowed
public long memoryConsumed()

Returns the number of bytes consumed so far in this memory area.

@SCJAllowed
public long memoryRemaining()

Returns the number of bytes remaining in this memory area.

@SCJAllowed
public Object newArray(**Class** type, **int** number)

Create an array object of class typeand length number in this memory area.

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 Object newInstance(**Class** type)

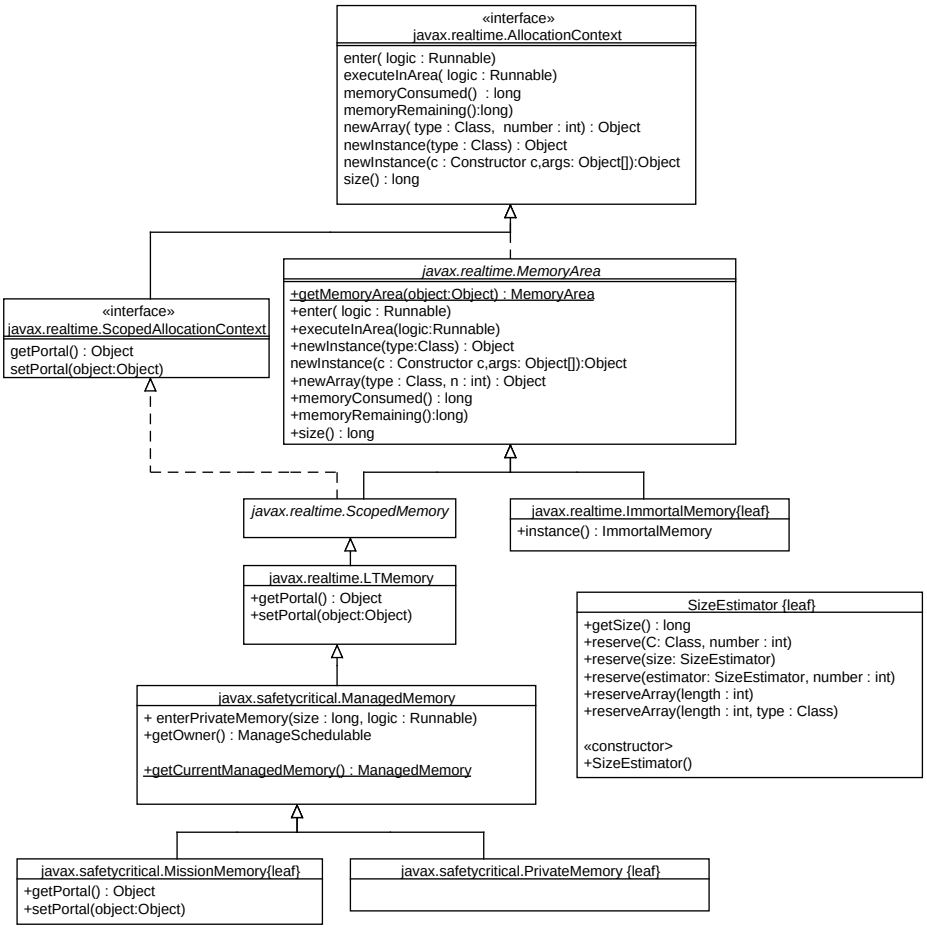

Figure 7.2: Overview of MemoryArea-Related Classes

Create an object of class type in this memory area.

Throws IllegalAccessException if the class or initializer is inaccessible.

Throws InstantiationException if the specified class object could not be instantiated.

Throws OutOfMemoryError if space in the memory area is exhausted.

@SCJAllowed
public Object newInstance(Constructor c, Object[] args)

Create an object of class type in this memory area.

Throws IllegalAccessException if the class or initializer is inaccessible.

Throws InstantiationException if the specified class object could not be instantiated.

Throws OutOfMemoryError if space in the memory area is exhausted.

@SCJAllowed
public long size()

Returns the current size in bytes of this memory area.

7.3.2 Interface javax.realtime.ScopedAllocationContext

Declaration

@SCJAllowed
public interface ScopedAllocationContext **extends** AllocationContext

A scoped allocation context is one whose associated objects have their memory reclaimed when no schedulable object is active within it.

Methods

@SCJAllowed
public Object getPortal()

Returns the portal object of this memory area.

@SCJAllowed
public void setPortal(Object value)

Sets the portal object of this memory area.

7.3.3 Class javax.realtime.MemoryArea

Declaration

@SCJAllowed
public abstract class MemoryArea **implements** AllocationContext

All allocation contexts are implemented by memory areas. This is the base-level class for all memory areas.

Static Methods

@SCJAllowed
public static MemoryArea getMemoryArea(Object object)

Returns the memory area in which object is allocated.

Methods

public void enter(Runnable logic);

Execute logic with this memory area as the current allocation context. Only the infrastructure calls this method directly.

@SCJAllowed
public void executeInArea(Runnable logic) **throws** InaccessibleAreaException

Execute logic with this memory area as the current allocation context. This memory area must already be active in the calling schedulable object's scope stack.

Throws java.lang.IllegalArgumentException when logic is null.

@SCJAllowed
public long memoryConsumed()

Returns the memory consumed in this memory area.

@SCJAllowed
public long memoryRemaining()

Returns the memory remaining in this memory area.

@SCJAllowed
public Object newInstance(**Class** type)

Returns a new object of class type allocated in this memory area.

Throws IllegalArgumentException, InstantiationException, OutOfMemoryError, InaccessibleAreaException

@SCJAllowed
public Object newInstance(Constructor c, Object[] args)

Create an object of class type in this memory area.

Throws IllegalAccessException if the class or initializer is inaccessible.

Throws InstantiationException if the specified class object could not be instantiated.

Throws OutOfMemoryError if space in the memory area is exhausted.

@SCJAllowed
public Object newArray(**Class** type, **int** n)

Returns a new array of type type and size n allocated in this memory area.

```
@SCJAllowed
public long size()
```

The size of a memory area is always equal to the memConsumed() + memoryRemaining().

Returns the total size of this memory area.

7.3.4 Class javax.realtime.ImmortalMemory

Declaration

```
@SCJAllowed
public final class ImmortalMemory extends MemoryArea
```

This class represents immortal memory. Objects allocated in immortal memory are never reclaimed during the lifetime of the application.

Static Methods

```
@SCJAllowed
public static ImmortalMemory instance()
```

Returns the singleton instance of the immortal memory class. The returned object is preallocated in immortal memory.

7.3.5 Class javax.realtime.ScopedMemory

Declaration

```
@SCJAllowed
public abstract class ScopedMemory extends MemoryArea
      implements ScopedAllocationContext
```

Scoped memory implements the scoped allocation context.

For the following reasons, this class has no visible methods in SCJ.

- In SCJ, all scoped memory areas are private to a schedulable object. Consequently, none of the RTSJ methods associated with sharing scoped memory areas are supported.
- The only sharable scoped memory area is the mission memory, and this behaves as if it is immortal memory for the lifetime of the mission.
- Only the infrastructure can resize a scoped memory area.

7.3.6 Class javax.realtime.LTMemory

Declaration

@SCJAllowed
public class LTMemory **extends** ScopedMemory

Description

This class can not be instantiated in SCJ. It is subclassed by MissionMemory and PrivateMemory. None of its methods are visible.

Methods

@SCJAllowed
public Object getPortal()

Returns the portal object.

@SCJAllowed
public void setPortal(Object value)

Set the portal object.

7.3.7 Class javax.safetycritical.ManagedMemory

Declaration

@SCJAllowed
public class ManagedMemory **extends** LTMemory

Description Managed memory is a scoped memory area that is managed by a mission

Static Methods

@SCJAllowed
public ManagedMemory getCurrentManageMemory()

Returns the current managed memory.

Throws IllegalStateException when called from immortal.

Methods

@SCJAllowed
public void enterPrivateMemory(**int** size, Runnable logic)

If private memory does not exist, create one; otherwise set its size; then, enter the private memory; and finally, set the size of private memory to zero.

Throws IllegalStateException when called from another memory area or from a thread that does not own the current managed memory.

@SCJAllowed
public ManagedSchedulable getOwner()

Returns the ManagedSchedulable that owns this managed memory.

7.3.8 Class javax.safetycritical.MissionMemory

Declaration

@SCJAllowed
public final class MissionMemory **extends** ManagedMemory

Description

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, non of its constructors are visible in the SCJ public API.

Methods

@SCJAllowed
public Object getPortal()

Returns the portal object.

@SCJAllowed
public void setPortal(Object value)

Set the portal object for sharing between managed schedulable objects.

7.3.9 Class javax.safetycritical.PrivateMemory

Declaration

@SCJAllowed
public final class PrivateMemory **extends** ManagedMemory

Description

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 static enterPrivateMemory method on 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.

7.3.10 Class javax.realtime.SizeEstimator

Declaration

```
@SCJAllowed
public final class SizeEstimator
```

Description

This class maintains an estimate of the amount of memory required to store a set of objects.

SizeEstimator is a floor on the amount of memory that should be allocated. 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 instance itself is allocated in several parts (if for instance 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 instance.

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.

Constructor

```
@SCJAllowed
public SizeEstimator()
```

Creates a new object in the current allocation context.

Methods

```
@SCJAllowed
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
public void reserve(Class c, int number)
```

Take into account additional number instances of Class c when estimating the size.

Parameter c is the class to take into account.

Parameter number is the number of instances of c to estimate.

Throws IllegalArgumentException if c is null.

```
@SCJAllowed
public void reserve(SizeEstimator estimator, int number)
```

Take into account additional number instances of SizeEstimator size when estimating the size.

Parameter estimator is the given instance of SizeEstimator.

Parameter number is the number of times to reserve the size denoted by estimator.

Throws IllegalArgumentException if estimator is null.

@SCJAllowed
public void reserve(SizeEstimator size)

Take into account an additional instance of SizeEstimator size when estimating the size.

Parameter size is the given instance of SizeEstimator.

Throws IllegalArgumentException if size is null.

@SCJAllowed
public void reserveArray(**int** length)

Take into account an additional instance of an array of length reference values.

Parameter length is the number of entries in the array.

Throws IllegalArgumentException if length is null.

@SCJAllowed
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.

Parameter length is the number of entries in the array.

Parameter type is the class representing a primitive type. The reservation will leave room for an array of length of the primitive type corresponding to type.

Throws IllegalArgumentException if length is negative, or type does not represent a primitive type.

7.4 Rationale

Traditionally, safety-critical applications allocate all their data structures before the execution phase of the applications begin. As a rule, they do not deallocate objects, since 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 is predicated on

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 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 quite simple. A single nesting structure is provided such that a given scoped memory can only be entered by a single thread at any given time and a scope may only be entered 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 objects in immortal memory are not reclaimable. Therefore, each application uses a global mission scope 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. Furthermore, it avoids 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 scoped memory during the execution phase of the application. All class initialization happens before the first initialization phase. Class objects are allocated in immortal memory, as defined in the RTSJ, see Chapter 3.

7.4.1 Nesting Scopes

In the simplest case, the memory of a mission memory is just mapped immortal memory, but when programmable restartability is required, a MissionMemory class is used. MissionMemory is nothing other than 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, except that it can be reinitialized at the end of each mission. All objects needed by the mission are allocated in the mission memory area by the runnable given to initialize the application. The area is exited only after all tasks have terminated. Optionally, some cleanup may be performed.

Since the MissionMemory is not cleared during the mission, it would be unsafe to allocate object in the MissionMemory during the execution of the mission; 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. Since PrivateMemory is based on LTMemory, this give a single level of nesting for the application.

The RTSJ provides for calling finalizers when the last thread exits a scoped memory. Since finalization can cause unpredictable delay, finalizers are not allowed in SCJ. The same effect can be obtained with try statement that includes a finally clause for any given scope.

In general, the SCJ conforms to the Java memory model. With respect to this memory model, AsyncEventHandlers behave like Java threads. Fields accessed from more than one AsyncEventHandler 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 AsyncEventHandlers are run in single thread context, synchronization should still be done to aid portability.

7.5 Compatibility

Chapter 8

Clocks, Timers, and Time

Many safety critical applications require precise timing mechanisms for maintaining real-time response. The 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 jitter and drift of all vendor-supplied clocks should be documented by the vendor.

8.1.1 Clocks

SCJ supports a single system real-time clock and user-defined clocks. As in the RTSJ, the real-time clock is monotonic and non-decreasing. The real-time clock in the RTSJ (queried through Clock.getRealtimeClock()) has an Epoch of January 1^{st}, 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 thereof. AbsoluteTime represents a specific point in time; RelativeTime represent a time interval.

8.1.3 RTSJ Constraints

Time triggered application code is constructed using periodic events (PeriodicEventHandler). The RTSJ classes OneShotTimer, PeriodicTimer, and Timer that can be used to schedule application logic in the RTSJ are not available in SCJ.

As java.util.Date is not part of the SCJ library, the related constructor in AbsoluteTime is omitted.

8.2 Level Considerations

As wait and notify are available only in SCJ level 2, the method waitForObject() from HighResolutionTime is available only at SCJ level 2.

8.3 API

Figure 8.1 gives an overview of the time related classes.

8.3.1 Class javax.realtime.Clock

Declaration

```
@SCJAllowed
public abstract class Clock
```

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.

User-defined clocks can be used for periodic event handlers, timed events, spin, and delay.

Constructor

```
@SCJAllowed
public Clock()
```

Methods

```
@SCJAllowed
public static Clock getRealtimeClock()
```

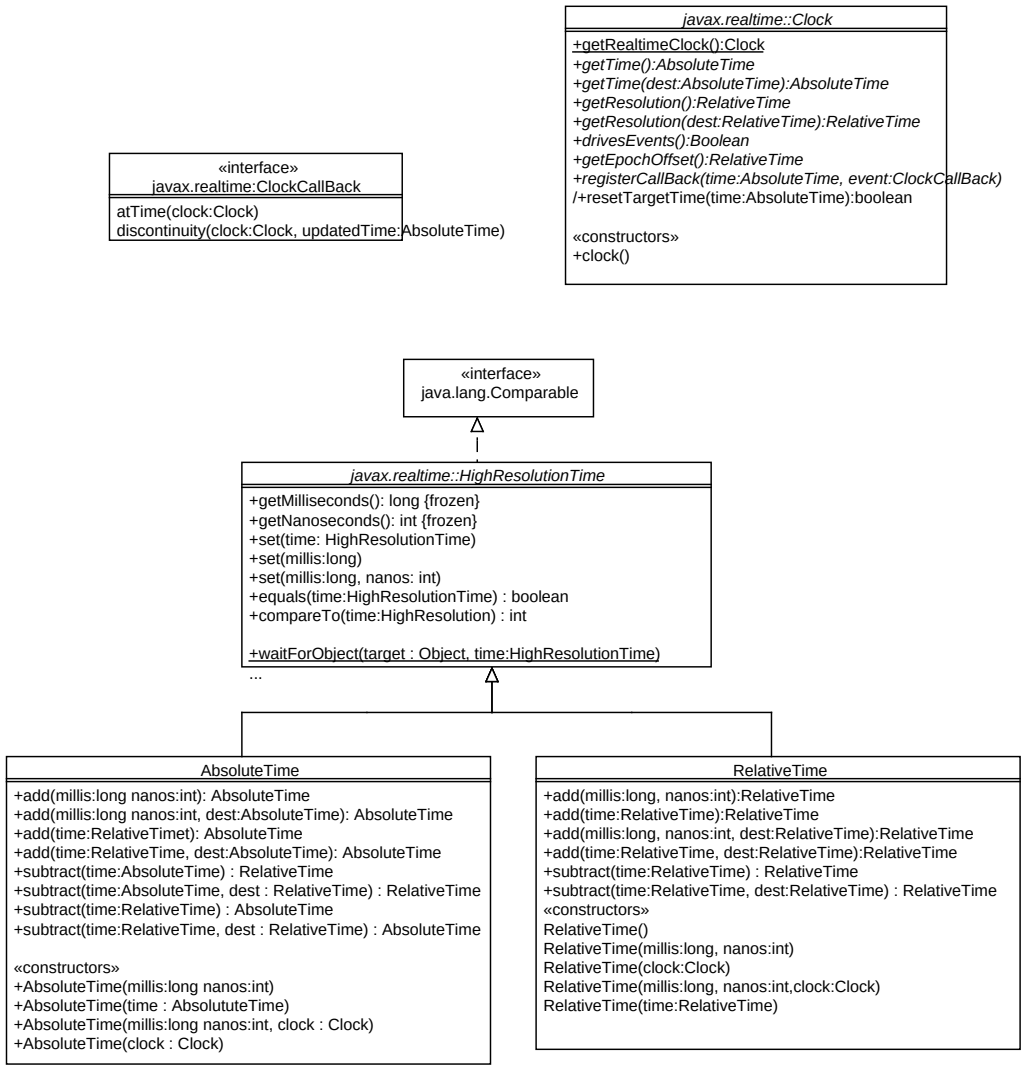

Figure 8.1: Abridged time classes

Returns the singleton instance of the default Clock.

@SCJAllowed
public abstract AbsoluteTime getTime()

Returns a newly allocated instance of AbsoluteTime in the current allocation context, which representing the current time. The returned object is associated with this clock. 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.

@SCJAllowed
public abstract 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 returns 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.

Parameter dest is 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
public abstract RelativeTime getResolution()

Returns a newly allocated RelativeTime object that indicates the nominal interval between ticks.

@SCJAllowed
public abstract RelativeTime getResolution(RelativeTime dest)

Gets the resolution of the clock, the nominal interval between ticks.

Parameter dest return the relative time value in dest. getTime with a destination null ignores it and return null.

Returns dest set to values representing the resolution of this. The returned object is associated with this clock.

@SCJAllowed
public abstract 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. A newly allocated RelativeTime object with the offset past the Epoch for this clock is returned. The returned object is associated with this clock.

Throws UnsupportedOperationException if the user-defined clock does not know its relation to the real-time clock.

@SCJAllowed
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() equal true is used to configure spin() request, an IllegalArgumentException will be thrown by the infrastructure.

@SCJAllowed(LEVEL_1)
protected abstract void registerCallBack(AbsoluteTime t, 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 clockEvent at time t, and if this clock is subject to discontinuities, invoke clockEvent.discontinuity(Clock, RelativeTime) each time a clock discontinuity is detected. Any already registered call backs are overridden.

This method behaves effectively as if it and invocations of clock events by this clock hold a common lock.

Parameter t is the absolute time value on this clock at which clockEvent.atTime(Clock) should be invoked.

Parameter clockEvent is the object that should be notified at time. If clockEvent is null, unregister the current clock event.

@SCJAllowed(LEVEL_1)
protected abstract boolean resetTargetTime(AbsoluteTime time)

Replace the target time being used by the ClockCallBack registered by registerCallBack(AbsoluteTime, ClockCallBack).

Returns false if no ClockCallBack event is currently registered.

8.3.2 Interface javax.realtime.ClockCallBack

Declaration

@SCJAllowed(LEVEL_1)
public interface ClockCallBack

Description

The ClockCallBack 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 for the same clock. The callback is de-registered before a method in it is invoked, and the clock blocks any attempt by another thread to register another callback while control is in a callback.

Methods

@SCJAllowed(LEVEL_1)
void atTime(Clock clock)

The clock has reached the designated time. This clock event is de-registered before this method is invoked.

Parameter clock is the clock that has reached a designated time.

@SCJAllowed(LEVEL_1)
void discontinuity(Clock clock, AbsoluteTime updatedTime)

The clock has experienced a time discontinuity. (It changed its time value other than by ticking.) and clock has de-registered this clock event.

Parameter updatedTime is the (new) current time.

8.3.3 Class javax.realtime.HighResolutionTime

Declaration

@SCJAllowed
public abstract class HighResolutionTime **implements** Comparable

The base class for AbsoluteTime and RelativeTime. Used to express time with nanosecond accuracy. This class is never used directly: it is abstract and has no public constructor.

Methods

@SCJAllowed
public Clock getClock()

Returns a reference to the clock associated with this.

@SCJAllowed
public final long getMilliseconds()

Returns the milliseconds component of the time represented by this.

@SCJAllowed
public final int getNanoseconds()

Returns the nanoseconds component of the time represented by this.

@SCJAllowed
public void set(HighResolutionTime time)

Change the value represented by this to that of the given time.

Parameter time is the new value for this.

@SCJAllowed
public void set(**long** millis)

Sets the millisecond component of this to the given argument, and the nanosecond component of this to 0.

Parameter millis is the value of the millisecond component of this at the completion of the call.

@SCJAllowed
public void set(**long** millis, **int** nanos)

Sets the millisecond and nanosecond components of this.

Parameter millis is the desired value for the millisecond component of this at the completion of the call. The actual value is the result of parameter normalization.

Parameter nanos is 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
public boolean equals(HighResolutionTime time)

Returns true if the parameter time is of the same type and has the same values as this.

@SCJAllowed
public int compareTo(HighResolutionTime time)

Compares this with the specified HighResolutionTime time.

@SCJAllowed(LEVEL_2)
public static void waitForObject(java.lang.Object target,
HighResolutionTime time) **throws** java.lang.InterruptedException

Behaves like wait() but with the enhancement that it waits with a precision of HighResolutionTime.

Parameter target is the object on which to wait. The current thread must have a lock on the object.

Parameter time is the time for which to wait. If it is RelativeTime(0,0) or null then wait indefinitely.

Throws java.lang.InterruptedException.

8.3.4 Class javax.realtime.AbsoluteTime

Declaration

@SCJAllowed
public class AbsoluteTime **extends** 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 operations, negative values behave as they do in arithmetic.

Constructors

@SCJAllowed
public AbsoluteTime(**long** millis, **int** nanos)

Construct an AbsoluteTime object with time millisecond and nanosecond components past the real-time clock's Epoch.

Parameter ms and ns are the desired values for the millisecond component and the nanosecond component, respectively, of this. The actual values are the result of parameter normalization.

@SCJAllowed
public AbsoluteTime(AbsoluteTime time)

Make a new object that is a copy from the parameter time.

@SCJAllowed
public AbsoluteTime(**long** millis, **int** nanos, Clock clock)

Construct an AbsoluteTime object with time millisecond and nanosecond components past the clock's Epoch.

Parameter ms and ns are the desired values for the millisecond component and the nanosecond component, respectively, of this. The actual values are the result of parameter normalization.

Parameter clock is the clock providing the association for the newly constructed object.

@SCJAllowed
public AbsoluteTime(Clock clock)

Equivalent to new AbsoluteTime(0,0,clock).

Methods

@SCJAllowed
public 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.

Parameter millis is the number of milliseconds to be added to this.

Parameter nanos is 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.

@SCJAllowed
public 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.

Parameter millis is he number of milliseconds to be added to this.

Parameter nanos is the number of nanoseconds to be added to this.

Parameter dest is the result is placed there and returned. If null, 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
public 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.

Parameter time is the time to add to this.

Returns a new object whose time is the normalization of this plus the parameter time.

@SCJAllowed
public 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.

Parameter time is the time to add to this.

Parameter dest is the result is placed there and returned. If null, 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
public 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.

Parameter time is the time to subtract from this.

Returns a new RelativeTime object whose time is the normalization of this minus the AbsoluteTime parameter time.

@SCJAllowed
public 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.

Parameter time is the time to subtract from this.

Parameter dest is the result is placed there and returned. If null, 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
public 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.

Parameter time is the time to subtract from this.

Returns a new AbsoluteTime object whose time is the normalization of this minus the parameter time.

@SCJAllowed
public 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.

Parameter time is the time to subtract from this.

Parameter dest is where the result is placed and returned. If null, 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.

8.3.5 Class javax.realtime.RelativeTime

Declaration

@SCJAllowed
public class RelativeTime **extends** HighResolutionTime

Description

An object that represents a time interval milliseconds/10^3 + nanoseconds/10^9 seconds long. It generally is used to represent a time relative to now.

Constructors

@SCJAllowed
public RelativeTime()

Equivalent to new RelativeTime(0,0).

@SCJAllowed
public RelativeTime(**long** ms, **int** ns)

Construct a RelativeTime object representing an interval based on the parameter millis plus the parameter nanos.

Parameters ms and ns are the desired values for the millisecond and nanosecond components of this. The actual values are the result of parameter normalization.

@SCJAllowed
public RelativeTime(Clock clock)

Equivalent to new RelativeTime(0, 0, clock).

Parameter clock is the clock providing the association for the newly constructed object.

@SCJAllowed
public RelativeTime(**long** ms, **int** ns, Clock clock)

Construct a RelativeTime object representing an interval based on the parameter ms plus the parameter nanos.

Parameters ms and ns are the desired values for the millisecond and nanosecond components of this. The actual values are the result of parameter normalization.

Parameter clock is the clock providing the association for the newly constructed object.

@SCJAllowed
public RelativeTime(RelativeTime time)

Make a new RelativeTime object from the given RelativeTime object.

Parameter time is the RelativeTime object which is the source for the copy.

Methods

@SCJAllowed
public 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.

Parameter millis is the number of milliseconds to be added to this.

Parameter nanos is 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.

@SCJAllowed
public 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.

Parameter time is the time to add to this.

Returns a new RelativeTime object whose time is the normalization of this plus millis and nanos.

@SCJAllowed
public 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.

Parameter millis is the number of milliseconds to be added to this.

Parameter nanos is the number of nanoseconds to be added to this.

Parameter dest is the result is placed there and returned. If null, 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
public 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.

Parameter time is the time to add to this.

Parameter dest is where the result is placed and returned. If null, 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
public 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.

Parameter time is the time to subtract from this.

Returns a new RelativeTime object whose time is the normalization of this minus the parameter time.

@SCJAllowed
public 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.

Parameter time is the time to subtract from this.

Parameter dest is where the result is placed and returned. If null, 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.

8.4 Rationale

A SCJ system may not have access to a time synchronization service. Therefore, SCJ does not require any particular *Epoch*. On a system without the notion of calender time AbsoluteTime(0,0) may represent the system startup time.

8.5 Compatibility

The RTSJ restriction on not using user-defined clock for scheduling does not apply for the SCJ runtime.

Chapter 9

Java Metadata Annotations

This chapter describes Java Metadata annotations used by the SCJ. Java Metadata annotations enable tool developers to add additional type-like information to a Java program to enable more detailed functional and non functional analyses, both for ensuring program consistency and for aiding the runtime system to produce more efficient code. The proposed 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 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.

The specification differentiate between *user code* and *infrastructure code*. User 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.

9.1 Semantics and Requirements

The SCJ annotations are addressing 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 in respect to specified SCJ level.
- *Memory Safety* – To ease certification and improve safety of developed software, SCJ dictates the memory management to be analyzable prior to system

annotation	parameters	description
@SCJAllowed	*level* = Level_0, Level_1, Level_2,	Specifies compliance level of an element.
	INFRASTRUCTURE,	An SCJ-library element that is API private.
	HIDDEN,	An element non-accessible both from user and infrastructure.
	members = TRUE / FALSE	if *TRUE* the annotation value is recursively inherited by sub-elements.

Figure 9.1: Compliance Levels Annotation

execution. A set of memory safety annotations is designed. For every object the area in which it is allocated must be statically determinable, allocation context of every method must be also know prior to execution.

- *Behavioral Restrictions* – Since the execution of the missions are 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 current execution phase of a mission.

9.1.1 Annotations for Enforcing Compliance Levels

API visibility annotations are used to prevent client programmers from accessing SCJ API methods that are intended to be internal. Since 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 which applications and implementations may conform to. 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 Figure 9.1 and takes two arguments:

1. The default argument of type Level specifies the level of the annotation target. The options are LEVEL_0, LEVEL_1, LEVEL_2, INFRASTRUCTURE and HIDDEN.
    - LEVEL 0, 1 or 2 specifies 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.
- 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 user 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 $<$ 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.

Compliance Level Reasoning

The compliance level of a class or member is the first of the following:

1. The level specified on its own @SCJAllowed() annotation, if it exists,
2. The level of the closest outer element with an @SCJAllowed() annotation, if members = true,
3. HIDDEN.

If a class, interface, or member has compliance level C, it may only 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. Intuitively, all of enclosed elements of a class or member should have a compliance level greater than or equal to the enclosing element.

Methods annotated HIDDEN or INFRASTRUCTURE may not be overridden in user code.

Static initializers have the same compliance level as their defining class, regardless of the members argument.

annotation	parameters	description
@SCJRestrict	MAY_BLOCK MAY_ALLOCATE ALLOCATE_FREE BLOCK_FREE	Specifies behavior of methods.
	INITIALIZATION ANY_TIME EXECUTION CLEANUP	Specifies mission context in which a certain method can be executed.

Figure 9.2: Annotation for Restricting Behavior

Class Constructor Rules

For a class that is annotated @SCJAllowed, all constructors have to be annotated @SCJAllowed as well.

If a class has a default constructor, the constructor's compliance level is that of the class if the annotation has members = true, or HIDDEN otherwise.

Other Rules

The exceptions thrown by a method must be visible at the compliance level of that method.

9.1.2 Annotations for Restricting Behavior

The following set of annotations is dedicated 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, see Figure 9.2.

The default argument is a list of restrictions of type Restrict. Supported restrictions and the intended semantics of these annotations are:

- MAY_ALLOCATE denotes a method which may allocate memory,
- ALLOCATE_FREE is annotated on methods that perform no allocation in themselves and only call methods that are also annotated @SCJRestricted(ALLOCATE_FREE). If a method is ALLOCATE_FREE, then all of its overrides are also considered to be ALLOCATE_FREE.
- BLOCK_FREE denotes a method which is guaranteed to not block,
- MAY_BLOCK denotes a method which may perform a blocking operation,
- INITIALIZATION denotes a method which can only be called during the initialization phase of a mission,

- CLEANUP denotes a method which may only be called during the clean-up phase,
- EXECUTION denotes a method which may be called during the mission execution phase.
- ANY_TIME denotes a method which may be called at any time.

When no annotations are present, the default value is MAY_BLOCK, MAY_ALLOCATE and ANY_TIME. Native methods have no defaults and thus must be fully annotated.

There is an ordering on annotations: MAY_BLOCK $<$ BLOCK_FREE, MAY_ALLOCATE $<$ ALLOCATE_FREE, INITIALIZATION $<$ ANY_TIME, EXECUTION $<$ ANY_TIME, CLEANUP $<$ ANY_TIME. A subclass is allowed to strengthen the annotations when overriding a method.

A method annotated ALLOCATE_FREE (respectively BLOCK_FREE) may only call methods that also have the ALLOCATE_FREE (respectively BLOCK_FREE) annotation.

A method annotated MAY_ALLOCATE (respectively MAY_BLOCK, INITIALIZATION and CLEANUP) can only be called from a method that also has the MAY_ALLOCATE (respectively MAY_BLOCK, INITIALIZATION and CLEANUP) annotation.

Only methods that are annotated MAY_ALLOCATE may contain expressions that result in allocation (e.g. at the source level new expressions, string concatenation, and autoboxing).

Methods can have multiple restriction annotations and must abide by the constraints of all of these restrictions. Annotations that are related by the ordering relation are mutually exclusive. INITIALIZATION and CLEANUP are mutually exclusive annotations.

Behavior Restriction Rules

Only methods annotated MAY_ALLOCATE may contain expressions that result in allocation. This includes new expressions, as well as string concatenation and autoboxing of primitive types to objects.

Methods annotated ALLOCATE_FREE may only invoke other methods that are annotated ALLOCATE_FREE. A similar rule is necessary for BLOCK_FREE. In addition, BLOCK_FREE methods may not have blocking statements, such as synchronized blocks.

Methods annotated INITIALIZATION, CLEANUP, or EXECUTION may only invoke other methods with the same annotation or ANY_TIME. Methods annotated ANY_TIME may only invoke other methods annotated ANY_TIME.

annotation	parameters	description
@DefineScope	*name* - name of the scope *parent* - parenting scope	Defines a scope and its parent.
@Scope	*name* - name of the scope where object lives	For an element specifies in which scope it is allocated.
@RunsIn	*name* - the scope where an element is executed.	Specifies allocation context of a method or class.

Figure 9.3: Memory Safety Annotations

9.1.3 Annotations for Memory Safety

The three SCJ annotations for memory safety, summarized in Figure 9.3, are:

- The @DefineScope annotation must 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 and B is the name of the direct ancestor of the scope. It is also used to annotate the Runnable passed to enterPrivateMemory to name the new scope being created.
- The annotation @Scope is added to class declarations to indicate that instances of the class are allocated in the named scope. 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.
- The annotation @RunsIn can be annotated on either classes or methods, which indicates an scope for the method or methods in question. When annotating a class, it signifies the default allocation context for its methods. When attached to a method, 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.

The special scope name Immortal is used to denote the singleton instance of ImmortalMemory. For each class that inherits an SCJ class that creates its own private memory (MissionSequencer, PEH, APEH), a scope is added to the scope tree with the name of the class as the name of the scope and the name specified by the @Scope annotation on that class as the parent scope.

@DefineScope Rules

For each @DefineScope, the variable that it is attached to must be declared in the allocation context specified by the parent argument to the annotation. Variable dec-

larations are checked to adhere to the following rules:

1. Every local variable or field annotated @DefineScope(name="A",parent="B") must be assigned a freshly created scoped memory object or be assigned a variable with the same annotation.

2. Variables of types annotated @Scope(a) are not allowed in any allocation context that is an ancestor of scope a in the scope hierarchy.

3. Every static field must have types that are annotated @Scope("Immortal") or are unannotated.

4. Lastly, Mission classes should also carry the @DefineScope annotation to name the mission memory and define what the parent scope of the defined mission is. It is not strictly necessary to annotate Mission objects; by default, a scope named after the mission is created, with immortal memory as the parent. Although event handlers also create implicit scoped memories, the necessary addition to the scope tree can be inferred from the @Scope and @RunsIn annotation on the event handler class.

All @DefineScope are checked to ensure that scope names are unique and that the parent relation forms a tree rooted at Immortal. During the first pass of checking, the scope tree is constructed and checked for well-formedness. That is, there can be no duplicates in the scope tree, or cycles. In addition, every chain of scopes must end at the immortal scope.

Object Allocation Rules

Rules for using objects annotated with the @Scope annotation are following:

- Objects may only be allocated in the context specified by the annotation on their types. This is the basis for all of the memory safety rules.
- Arrays may only be allocated in the same context as that of their element type.
- Variables of a specific type may not be declared in any scope that is a parent of the scope specified by the type. Intuitively, if all objects are allocated in a particular scope, a reference from a parent scope will result in a dangling pointer.
- Static variables must have types with no @Scope annotation or @Scope("immortal"). This is because static variables never leave scope during the lifetime of a program. Furthermore, no method running in any scope can store into a static reference variable.

- Child classes inherit the scope of parent classes. However, if parent class has no scope annotation, the child class may define the @Scope annotation arbitrarily.
- Each Schedulable class must have @Scope(”x”) and @RunsIn(”y”) annotations, defining that the object is allocated in scope x while it is running in scope y.

Class casting must follow special rules in order to not to lose the scope knowledge associated with each variable. Therefore for the example below:

```
@Scope(”x”)
class A {...}

@Scope(”y”)
class B extends A {...}

A a = (A) b;
```

the following rules must apply. Either scope x == y or, in case of x == null, the scope y must be equal to the current allocation context, otherwise a class-cast error must be reported.

Method Allocation Context

The allocation context of a method is the first of:

1. s, if the method is annotated @RunsIn(s)
2. s, if the class that the method belongs to is annotated @RunsIn(s)
3. s, if the class that the method belongs to is annotated @Scope(s)

Furthermore, methods inherit any annotations from methods that they override and are not allowed to change them.

Each method invocation must be checked for the following:

1. Invocation of a method is only allowed if its allocation context is the same as the current context or is a parent to it and is allocation free. This is simply to prevent objects from being allocated in the wrong scope.
2. The @Scope and @RunsIn annotations together define the allocation context for each method in an SCJ program. Annotations on methods take precedence over annotations on classes, with @RunsIn taking precedence over @Scope.

3. Calls to a scope's executeInArea() method can only be made if the scoped memory is a parent of the current context in the scope tree. In addition, the Runnable object passed to the method must have a RunsIn annotation that matches the name of the scoped memory. As with RTSJ, executeInArea() is used to execute some logic in a scope below the current scope on the scope stack.

4. Calls to a scope's enterPrivateMemory() method are only valid if the annotations are matching. In particular, the parent argument of the @DefineScope on the Runnable must be the same as the allocation context of the call site. The @RunsIn annotation of the Runnable must be the name of the scope being defined by @DefineScope(name=n, parent=m). In addition when invoking enter(), the current allocation context must be m In other words, m must be below n on the scope stack. On the other hand, when invoking executeInArea(), the n must be below the current allocation context in the scope stack.

5. If the method is MemoryArea.newInstance(), the created type must be completely unannotated or have be annotated @Scope(n), where n is the name of the scope upon which newInstance() is called. If the method is MemoryArea.newArray(), a similar rule applies with the base element type.

6. When invoking a method with parameters of unannotated types, the method must have the same allocation context as the current one.

7. When invoking a method where all the parameter types are annotated, the method must have the same allocation context as the current one or must be a parent allocation context and is Alloc restricted.

Methods that have allocation context s can only allocate objects whose types are annotated @Scope(s) or is completely unannotated. Likewise, they can only allocate arrays whose base element type is annotated @Scope(s) or is completely unannotated.

If a method has a return type that is unannotated, it is required that the object that is returned be allocated in the allocation context of the method.

In a method of an object annotated @Scope("A"), the valid operations on a reference of a class annotated @Scope("B"), where B is a parent of A, are reading and writing of primitive references, reading or writing of references annotated @Scope and invocation of methods annotated @AllocFree.

Rules for Unannotated Classes

In order to handle object of unannotated types (types with no memory-related annotations), we augment the above set of rules:

- Objects of unannotated types may be allocated anywhere. Since we can statically determine the allocation context in any annotated class, the allocation context of an unannotated object can subsequently be determined.
- Objects of unannotated types and, recursively, all of their fields of unannotated types are assumed to reside in the scope in which the root object is declared. All of its methods must also run in the same scope.
- Objects of unannotated types may not leave the allocation context in which they were instantiated (i.e., may not be passed to a method which has a different allocation context). Allowing an unannotated object to pass to a different scope from the one in which it was created would lose scope information necessary for determining assignability.
- When a method returns an object of an unannotated type, it must be allocated in the method's allocation context.
- Classes that are unannotated may not reference any annotated types. This prevents objects of unannotated types, which can be allocated anywhere, from allocating objects in a scope other than their designated one. In essence, unannotated code can be thought of as library code. Therefore, there is no reason why the library have knowledge of client code.

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.

9.2 Level Considerations

These annotations apply to all levels.

9.3 API

9.3.1 Class javax.safetycritical.annotate.SCJRestricted

Declaration

```
@Retention(CLASS)
@Target( { TYPE, FIELD, METHOD, CONSTRUCTOR })
```

public @interface SCJRestricted

Methods

public Restrict[] value() **default** {MAY_BLOCK, MAY_ALLOCATE, ANY_TIME}

Declaration

This annotation distinguishes methods that may be called only from a certain context (e.g. cleanup) or method that may be restricted to execute no memory allocation or blocking.

9.3.2 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 levels 0, 1, or 2. The value field equals LEVEL_1 for elements that may be used within safety-critical Java applications targeting levels 1 or 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

9.3.3 Class javax.safetycritical.annotate.Level

Declaration

public enum Level

LEVEL_0
LEVEL_1
LEVEL_2
INFRASTRUCTURE
HIDDEN

Description

Provides a set of possible values for the @SCJAllowed annotation's argument *level*.

9.3.4 Class javax.safetycritical.annotate.Restrict

Declaration

public enum Restrict

MAY_ALLOCATE
MAY_BLOCK
BLOCK_FREE
ALLOCATE_FREE
INITIALIZE
CLEANUP
ANY_TIME

Description

Provides a set of possible values for the @SCJRestricted annotation value.

9.4 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. The virtual machine may omit memory access checks for classes that have been successfully checked.

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. Since 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 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).

9.4.1 Compliance Level Annotation Example

The following example illustrates application of the compliance level annotation on a simple example. The example shows both user 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;

  public void initialize() {
   peh = new MyHandler(...);          // ERROR
   peh.run();                 // ERROR
  }
}
```

As we can see, all the elements of the example are declared to reside in a specific compliance level. At the user domain, we define class MyMission that 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.

```
@SCJAllowed (LEVEL_0)
public interface Schedulable
                        extends Runnable {
 @SCJAllowed(LEVEL_2)
 public ReleaseParameters getReleaseParameters();
}

@SCJAllowed(LEVEL_1)
class MyHandler extends
      PeriodicEventHandler {

  public void handleEvent() {
      ...
  }
}

@SCJAllowed(LEVEL_0)
public abstract class
        PeriodicEventHandler
        extends ManagedEventHandler
        implements Runnable {

  @SCJAllowed(LEVEL_0)
  public PeriodicEventHandler(..) {
         super(...);
  }

  @SCJAllowed(LEVEL_0)    // ERROR
  public ReleaseParameters
         getReleaseParameters() {
         ...
  }
  @SCJAllowed(INFRASTRUCTURE)
  public final void run() {
      ...
  }
}
```

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.

9.4.2 Memory Safety Annotations Example

The following user-level code snippet illustrated application of memory safety annotations. The example shows a user-domain source code fragment that defines a MyMission class, where we explicitly declare a scope in which the mission is running by @DefineScope(name="MyMission",parent="immortal"). Furthermore, mission's handler MyHandler is defined to be allocated in mission's memory by @Scope("MyMission"), while running in its own handler's private memory by @RunsIn("MyHandler"), thus implicitly defining a new memory area.

```
@Scope("immortal")
@DefineScope(name="MyMission",
                    parent="immortal")
class MyMission extends CyclicExecutive {
    public void initialize() {
        new MyHandler(...);
    }
}

@Scope("MyMission")
@RunsIn("MyHandler")
class MyHandler extends PeriodicEventHandler {

  public void handleEvent() {
      ManagedMemory.getCurrentManagedMemory().
          enterPrivateMemory(3000, new
            /*@DefineScope(name="MyTestRunnable",
                          parent="MyHandler")*/
            MyTestRunnable());
  }
}

@Scope("MyHandler")
@RunsIn("MyTestRunnable")
class MyTestRunnable implements Runnable {
     public void run() {
     }
}
```

The user is also expected to define a new scope area any time code enters a child scope. This is illustrated by the MyTestRunnable class that is allocated in MyHandler private memory while running in its own scope. Note, the specific approach to annotation of the enterPrivateMemory. Since the Java annotation system does not allow to annotate method invocation, we place the annotation upon the instance of the MyTestRunnable enclosed by the comments . Furthermore, notice that the memory areas form a tree with the immortal scope in root.

9.4.3 A Large-Scale Example

This final large-scale example present a typical SCJ level 0 application with all three types of SCJ annotations. On a simple user-level code snippet, we implement a level 0 mission – MyMission and its periodic handler – MyHandler. The MyMission object 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 mission's handler and then uses enterPrivateMemory() to perform some initialization tasks in sub-scopes using ARunnable and BRunnable.

```
package mission;

@SCJAllowed(LEVEL_0)
@Scope("MyMission")
class MyMission extends Mission {

  static Immortal instance = new Immortal();
  A a = new A();          // ERROR


  @SCJRestricted({MAY_ALLOCATE, MAY_BLOCK, INITIALIZATION})
  void initialize() {
  B aObj = new B();
  handler.B bObj = new handler.B();       // ERROR
  new MyHandler();

  ManagedMemory.getCurrentManagedMemory().
        enterPrivateMemory(1000, new
        /*@DefineScope(name="MyMissionInit", parent="MyMission")*/
        ARunnable());          // ERROR

   ManagedMemory.getCurrentManagedMemory().
        enterPrivateMemory(1000, new
        /*@DefineScope(name="PrivateMemory", parent="MyMission")*/
        BRunnable());          //ERROR
 }

@SCJAllowed(LEVEL_1)
@Scope("MyMission")
@RunsIn("MyMissionInit")
class ARunnable implements Runnable { ... }

@SCJAllowed(LEVEL_0)
@Scope("MyHandler")
@RunsIn("BRunnable")
class BRunnable implements Runnable { ... }

@SCJAllowed(LEVEL_0)
@Scope("MyMission")
```

class B { ... }

The figure also highlights several errors that would be detected by the checker. First, the user attempts to allocate instances of classes A and B in an allocation context that is not consistent with those defined for these classes. In order to instantiate properly the class B in MyMission class, the user has not other choice then to duplicate the class implementation and provide each class declaration with a different scope annotation, as it is shown in the example where B is defined in both packages with corresponding scope annotations.

Furthermore, the usage of both handlers is also illegal. The ARunnable class is annotated to be level 1 and therefore is not accessible in the context of the level 0 MyMission. The BRunnable class is defined to run in the BRunnable scope; however, the initialization method defines the instance of BRunnable class to run in Private-Memory.

```
package handler;

@SCJAllowed(value=LEVEL_0, members=true)
@Scope("MyMission")
@RunsIn("MyHandler")
class MyHandler extends PeriodicEventHandler {
   @SCJRestricted({MAY_ALLOCATE,MAY_BLOCK})
   void handleEvent() {
      A aObj = new A();          // ERROR
      B bObj = new B();
      @DefineScope(name="MyHandler", parent="MyMission")
      ManagedMemory mem = ManagedMemory.getCurrentManagedMemory();

      mem.enterPrivateMemory(1000, new
         /*@DefineScope(name="MyMissionInit", parent="MyHandler")*/
         ARunnable());          // ERROR

      mem.enterPrivateMemory(1000, new
         /*@DefineScope(name="BRunnable", parent="MyHandler")*/
         BRunnable());
    }
}

@SCJAllowed(LEVEL_0)
@Scope("MyMission")
class A {
   @SCJRestricted({MAY_ALLOCATE})
   void bar() { }
}

@SCJAllowed(LEVEL_0)
@Scope("MyHandler")
class B {
```

```
    A a;
    A a2 = new A();          // ERROR
    Object o;

    @SCJRestricted({ALLOCATE_FREE})
    void foo(A a) {
        a.bar();             // ERROR
        o = a;               // ERROR
    }
}
```

The MyHandler class implements functionality – the handleEvent() method – that will be periodically executed throughout the mission. The allocation context of this execution will be MyHandler scope, as the RunsIn annotation upon the MyHandler class suggests. Looking at the handleEvent() method, we can see that some of the functionality is designated to be executed in child scope memories through the ARunnable and BRunnable classes. However, the checker will detect the error since the ARunnable is declared to run in the MyMissionInit memory area, which is not a child to the MyHandler memory area. Furthermore, the MyHandler and B classes try to instantiate the A class in illegal contexts. Method bar() in B further tries to call a.foo(), which is annotated as MAY_ALLOCATE whereas the caller method prohibits any allocation. Finally, the assignment o = a; is illegal, since the user is assigning a variable from scope MyMission to a variable MyHandler. Assigning an instance of an unannotated type into a variable in a different scope would cause lost of scope knowledge related to this instance, which is illegal.

Chapter 10

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. Levels 1 and 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 0, 1, and 2 levels of SCJ, vendors may offer additional library support to complement the core capabilities.

See the javadoc sections 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

10.1 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.

10.2 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 was agreed 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.

TBD: James and Jan say no Iterator - Martin says this should be present if collections are present. I believe we wanted to leave this issue "open" until consensus is achieved.

`Readable` interface present in JDK 1.6 is not included in 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 ommissions was to reduce the size and complexity of the `Character` class. The judgement of the SCJ expert group was that safety-critical code would generally not be involved with text processing.

The class `Character.Subset` is omitted from the SCJ specification. The rationale for this ommission was to reduce the size and complexity of the `java.lang` package. The judgement of the SCJ expert group was that safety-critical code would generally not be involved with text processing.

The class `Character.UnicodeBlock` is omitted from the SCJ. The rationale for this ommission was to reduce the size and complexity of the `java.lang package`. The judgement of the SCJ expert group was that safety-critical code would generally not be involved with 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 ommissions was to reduce the size and complexity of the Class class. The expert group decided to severely restrict reflection.

Note that `Class` class does implement the following methods:

```
getEnumConstants(),
getSuperclass(),
gisAnnotation(),
gisArray(),
gisAssignableFrom(Class),
gisEnum(),
gisInstance(Object),
gisInterface(),
gnewInstance(),
gtoString().
```

TBD: There may be some further discussion required to determine whether all of these methods are in the SCJ specification.

The class `ClassLoader` is omitted from the SCJ. The expert group decided that dynamic class loading would 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. The expert group expects 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 same as JDK 1.6.

The class `Enum`: SCJ specification is 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 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 same as JDK 1.6.

The class `Long`: SCJ specification is same as JDK 1.6.

The class `Math`: SCJ specification is same as JDK 1.6.

The class `Number`: SCJ specification is 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.

TBD: Does the Purdue consistency checker enforce this "meaning" of not @SCJAllowed?

The following methods:

`notify()`,
`notifyAll()`,
`wait()`,
`wait(long timeout)`,
and `wait(long timeout, int nanos)`

are only `@SCJAllowed` at level 2. The SCJ expert group chose to limit the use of these services in order to enable a simpler run-time environment and easier analysis of real-time schedulability in levels 0 and 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 same as JDK 1.6.

The class `StackTraceElement`: SCJ specification is same as JDK 1.6.

The class `StrictMath`: SCJ specification is 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 ommissions was to reduce the size and complexity of the String class. In general, the SCJ expert group judged 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 ommissions was 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`. In general, the SCJ expert group judged 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 ommissions was 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 ommissions was 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 ommissions was to reduce the size and complexity of the `Throwable` class and subclasses.

The class `ArithmeticException`: SCJ specification is same as JDK 1.6.

The class `ArrayIndexOutOfBoundsException`: SCJ specification is same as JDK 1.6.

The class `ArrayStoreException`: SCJ specification is same as JDK 1.6.

The class `ClassCastException`: SCJ specification is same as JDK 1.6.

The class `ClassNotFoundException`: SCJ specification is same as JDK 1.6.

The class `CloneNotSupportedException`: SCJ specification is 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 same as JDK 1.6.

The class `IllegalThreadStateException` is omitted from the SCJ specification because it is not needed.

The class `IndexOutOfBoundsException`: SCJ specification is same as JDK 1.6.

The class `InstantiationException`: SCJ specification is same as JDK 1.6.

The class `InterruptedException`: SCJ specification is same as JDK 1.6.

The class `NegativeArraySizeException`: SCJ specification is 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 same as JDK 1.6.

The class `NumberFormatException`: SCJ specification is same as JDK 1.6.

The class `RuntimeException`: SCJ specification is 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 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 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 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 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 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 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 same as JDK 1.6.

The class `StackOverflowError`: SCJ specification is 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 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 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.

10.3 Comparison of SCJ API with JDK 1.6 java.lang.annotation

The interface `Annotation`: SCJ specification is 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 same as JDK 1.6.

The class `Inherited`: SCJ specification is same as JDK 1.6.

The class `Retention`: SCJ specification is same as JDK 1.6.

The class `Target`: SCJ specification is same as JDK 1.6.

10.4 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.

TBD: as a group, we don't yet have consensus on whether or not to include Iterator support.

Chapter 11

JNI

11.1 Semantics and Requirements

The RTSJ provides only minimal restrictions on calls to native interfaces. This chapter defines the additional restrictions that are required for SCJ. If the underlying system supports native code execution, then all JNI supported services shall be implemented; otherwise, JNI is not available to the application.

11.2 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, 1 and 2.

11.3 API

11.3.1 Supported Services

These JNI services in this section are supported by each SCJ implementation.

General service to get JNI version information:

- GetVersion

General object analysis: The following methods provide basic operation 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 operation on strings and require no reflection or other hard-to-analyze code.

- GetStringLength
- GetStringUTFLength
- GetStringRegion
- GetStringUTFRegion

Array Operations The following methods provide basic operation 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 may only be called 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

11.3.2 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 programmer considers hers or his 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.

As usual, an annotation with @SCJAllowed() is also required for each native method.

11.4 Rationale

Due to the complexity of static analysis of code that contains reflection, the SCJ restricts all uses 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 since the corresponding operations are not supported.

11.4.1 Unsupported Services

The 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 re-defined 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 since 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 since it is only useful for systems with dynamic loading:

- UnregisterNatives

The following memory management services are NOT required to be supported since 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 since 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

11.5 Example

```
@SCJAllowed
@ALLOCATE_FREE
@MAY_BLOCK
static native int getProcessorId(String theProcessorInformationString);
```

The native method is called with a previously allocated string as parameter. Beside 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. Since 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 **Please insert reference to:**

http://java.sun.com/javase/6/docs/technotes/guides/jni/ obeying the restrictions of the previous section.

11.6 Compatibility

11.6.1 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 fine under a less restricted environment like 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 facultativly allocation of Java objects from native code, such allocations should allocate objects using the current allocation context at the point of the call.

11.6.2 General Java Compatibility Issues

Existing JNI code may need to be modified, 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 in as an argument to the underlying JNI function because there is no way to access a field directly.

Chapter 12

Exceptions

Exceptions are normally considered a good mechanism to separate 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 analyse.

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 data flow analysis can go a long way to determine 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 catch 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.

12.1 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 that of RTSJ.

In SCJ, every Schedulable is 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 for a description of one possible approach.

12.1.1 New 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 dependent.

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 adds a number of new exceptions, thrown at runtime, when assignment rules between memory areas are violated. To avoid those exceptions, Chapter 9 introduces annotations for scope safe SCJ programs. Correctly annotated programs are guaranteed to never throw any of the scope related exceptions (IllegalAccessException, ScopedCycleException, InaccessibleAreaException).

12.2 Level Considerations

The support for exceptions is the same for all levels. A method annotated with a particular compliance level shall neither declare nor throw exceptions which have a higher compliance level.

12.3 API

The classes Error and Exception in java.lang provide the same constructors and methods in SCJ and in standard Java. The class Throwable in java.lang provides the same constructors in SCJ and in standard Java; the available methods are restricted in SCJ as described below.

12.3.1 Class java.lang.Error

Declaration

```
@SCJAllowed
public class Error extends Throwable implements Serializable
```

Constructors

```
@Allocate({CURRENT})
@BlockFree
@SCJAllowed
public Error()
```

Invokes System.captureStackBacktrace(this) to save the back trace associated with the current thread.

```
@Allocate({CURRENT})
@BlockFree
@MemoryAreaEncloses(inner = {"this"}, outer = {"message"})
@SCJAllowed
public Error(String message)
```

Invokes System.captureStackBacktrace(this) to save the back trace associated with the current thread.

```
@Allocate({CURRENT})
@BlockFree
@MemoryAreaEncloses(inner = {"this", "this"}, outer = {"message", "cause"})
@SCJAllowed
public Error(String message, Throwable cause)
```

Does not invoke System.captureStackBacktrace(this) so as to not overwrite the back-trace associated with cause.

```
@Allocate({CURRENT})
@BlockFree
@MemoryAreaEncloses(inner = {"this"}, outer = {"cause"})
@SCJAllowed
public Error(Throwable cause)
```

Does not invoke System.captureStackBacktrace(this) so as to not overwrite the back-trace associated with cause.

12.3.2 Class java.lang.Exception

Declaration

```
@SCJAllowed
public class Exception extends Throwable implements Serializable
```

Constructors

```
@Allocate({CURRENT})
@BlockFree
@SCJAllowed
public Exception()
```

Invokes System.captureStackBacktrace(this) to save the back trace associated with the current thread.

```
@Allocate({CURRENT})
@BlockFree
@MemoryAreaEncloses(inner = {"this"}, outer = {"message"})
@SCJAllowed
public Exception(String message)
```

Invokes System.captureStackBacktrace(this) to save the back trace associated with the current thread.

```
@Allocate({CURRENT})
@BlockFree
@MemoryAreaEncloses(inner = {"this", "this"}, outer = {"message", "cause"})
@SCJAllowed
public Exception(String message, Throwable cause)
```

Does not invoke System.captureStackBacktrace(this) so as to not overwrite the back-trace associated with cause.

```
@Allocate({CURRENT})
@BlockFree
@MemoryAreaEncloses(inner = {"this"}, outer = {"cause"})
@SCJAllowed
public Exception(Throwable cause)
```

Does not invoke System.captureStackBacktrace(this) so as to not overwrite the back-trace associated with cause.

12.3.3 Class java.lang.Throwable

Declaration

```
@SCJAllowed
public class Throwable implements Serializable
```

Constructors

```
@Allocate({CURRENT})
@BlockFree
@SCJAllowed
public Throwable()
```

Invokes System.captureStackBacktrace(this) to save the back trace associated with the current thread.

```
@Allocate({CURRENT})
@BlockFree
@MemoryAreaEncloses(inner = {"this"}, outer = {"message"})
@SCJAllowed
public Throwable(String message)
```

Invokes System.captureStackBacktrace(this) to save the back trace associated with the current thread.

```
@Allocate({CURRENT})
@BlockFree
@MemoryAreaEncloses(inner = {"this", "this"}, outer = {"message", "cause"})
@SCJAllowed
public Throwable(String message, Throwable cause)
```

Does not invoke System.captureStackBacktrace(this) so as to not overwrite the back-trace associated with cause.

```
@Allocate({CURRENT})
@BlockFree
@MemoryAreaEncloses(inner = {"this"}, outer = {"cause"})
@SCJAllowed
public Throwable(Throwable cause)
```

Does not invoke System.captureStackBacktrace(this) so as to not overwrite the back-trace associated with cause.

Methods

```
@BlockFree
@SCJAllowed
public String getMessage()
```

Performs no memory allocation. 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.

@BlockFree
@SCJAllowed
public Throwable getCause()

Performs no memory allocation. 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.

@Allocate({CURRENT})
@BlockFree
@SCJAllowed
public StackTraceElement[] getStackTrace() **throws** IllegalStateException

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.

12.3.4 Class jaxax.safetycritical.ThrowBoundaryError

Declaration

@SCJAllowed
public class ThrowBoundaryError **extends** javax.realtime.ThrowBoundaryError

Constructors

@SCJAllowed
public ThrowBoundaryError()

Allocates an application- and implementation-defined amount of memory in the current scope (to represent stack backtrace).

Methods

@SCJAllowed
public String getPropagatedMessage()

Returns Allocates and returns a 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.

@SCJAllowed
public StackTraceElement[] getPropagatedStackTrace()

Returns Allocates and returns 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 ThrowBoundaryError object.

@SCJAllowed

public int getPropagatedStackTraceDepth()

Returns the number of valid elements stored within the StackTraceElement array to be returned by getPropagatedStackTrace().

@SCJAllowed
public Class getPropagatedExceptionClass()

Returns a reference to the Class of the exception most recently thrown across a scope boundary by the current thread.

12.4 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 compliant SCJ 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 into a thread-local variable a reference to its Throwable, using some virtual machine mechanism if necessary, to avoid throwing an IllegalAssignmentError. At a subsequent invocation of Throwable.getStackTrace(), the 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 StackTrace-

Element 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 virtual machine (such as ArithmeticException, OutOfMemoryError, StackOverflowError) are preallocated in immortal memory. Immediately before throwing a preallocated exception, the virtual machine infrastructure invokes Services.captureBackTrace() to overwrite the stack back trace associated within the current thread with the preallocated exception.

SCJ requires exceptions to be immutable to ensure that repeated throwing of one exception object does not cause a memory leak.

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.

12.5 Compatibility

12.5.1 RTSJ Compatibility Issues

The precise semantic 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 errors even if the exception is propagated through more scopes. Existing RTSJ code which is sensitive to the origin of ThrowBoundaryError will require changes.

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.

12.5.2 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.

Appendix A

Javadoc Description of Package java.io

Package Contents *Page*

Interfaces

Closeable 189

Unless specified to the contrary, see JDK 1.

Flushable 189

Unless specified to the contrary, see JDK 1.

Serializable 189

This interface is provided for compatibility with standard edition Java.

Classes

FilterOutputStream 190

Unless specified to the contrary, see JDK 1.

IOException 191

...no description...

InputStream 192

Unless specified to the contrary, see JDK 1.

OutputStream 193

Unless specified to the contrary, see JDK 1.

A.1 Interfaces

A.1.1 INTERFACE Closeable

Unless specified to the contrary, see JDK 1.6 documentation.

DECLARATION

```
@SCJAllowed
public interface Closeable
```

METHODS

```
@SCJAllowed
public void close( )
```

A.1.2 INTERFACE Flushable

Unless specified to the contrary, see JDK 1.6 documentation.

DECLARATION

```
@SCJAllowed
public interface Flushable
```

METHODS

```
@SCJAllowed
public void flush( )
```

A.1.3 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.

DECLARATION

```
@SCJAllowed
public interface Serializable
```

A.2 Classes

A.2.1 CLASS FilterOutputStream

Unless specified to the contrary, see JDK 1.6 documentation.

DECLARATION

```
@SCJAllowed
public class FilterOutputStream
  extends java.io.OutputStream
```

CONSTRUCTORS

```
@SCJAllowed
@BlockFree
@MemoryAreaEncloses(inner = {"this"}, outer = {"out"})
public FilterOutputStream( OutputStream out )
```

METHODS

```
@SCJAllowed
public void  close( )
```

```
@SCJAllowed
public void  flush( )
```

```
@SCJAllowed
public void  write( byte []b )
```

```
@SCJAllowed
public void  write( byte []b , int off , int len )


@SCJAllowed
public void  write( int b )
```

A.2.2 CLASS **IOException**

DECLARATION

```
@SCJAllowed
public class IOException
  implements java.io.Serializable
  extends java.lang.Exception
```

CONSTRUCTORS

```
@BlockFree
@SCJAllowed
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.

```
@BlockFree
@MemoryAreaEncloses(inner = {”this”}, outer = {”msg”})
@SCJAllowed
public IOException( String msg )
```

Shall not copy ”this” to any instance or static field. The scope containing the msg argument must enclose the scope containing ”this”. Otherwise, an IllegalAssignmentError will be thrown.

Invokes System.captureStackBacktrace(this) to save the back trace associated with the current thread.

A.2.3 CLASS **InputStream**

Unless specified to the contrary, see JDK 1.6 documentation.

DECLARATION

```
@SCJAllowed
public abstract class InputStream
  implements java.io.Closeable
  extends java.lang.Object
```

CONSTRUCTORS

```
@BlockFree
@SCJAllowed
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 )
```

```
@SCJAllowed
public int  read( byte []b , int off , int len )


@SCJAllowed
public abstract int  read( )


@SCJAllowed
public void  reset( )


@SCJAllowed
public long  skip( long n )
```

A.2.4 CLASS OutputStream

Unless specified to the contrary, see JDK 1.6 documentation.

DECLARATION

```
@SCJAllowed
public abstract class OutputStream
  implements java.io.Closeable, java.io.Flushable
  extends java.lang.Object
```

CONSTRUCTORS

```
@BlockFree
@SCJAllowed
public OutputStream( )
```

METHODS

```
@SCJAllowed
public void  close( )
```

@SCJAllowed
public void **flush**( )

@SCJAllowed
public void **write**(byte []b)

@SCJAllowed
public void **write**(byte []b , int off , int len)

@SCJAllowed
public abstract void **write**(int b)

Appendix B

Javadoc Description of Package java.lang

Package Contents *Page*

Interfaces

Appendable 202

...no description...

CharSequence 202

...no description...

Cloneable 203

...no description...

Comparable 204

...no description...

Deprecated 204

...no description...

Override 204

...no description...

Runnable 205

...no description...

SuppressWarnings 205

...no description...

Thread.UncaughtExceptionHandler 205

...no description...

Classes

ArithmeticException 206

...no description...

ArrayIndexOutOfBoundsException 207

...no description...

ArrayStoreException 208

...no description...

AssertionError 209

...no description...

BigDecimal 211

...no description...

BigInteger 223

...no description...

Boolean 232

...no description...

Byte 235

...no description...

Character 240

...no description...

Class 247

...no description...

ClassCastException 250

...no description...

ClassNotFoundException 251

...no description...

CloneNotSupportedException 252

...no description...

Double 253

...no description...

Enum 259

...no description...

Error 261

...no description...

Exception 263

...no description...

ExceptionInInitializerError 264

...no description...

Float 265

...no description...

IllegalArgumentException 271

...no description...

IllegalMonitorStateException 273

...no description...

IllegalStateException 274

...no description...

IllegalThreadStateException 275

...no description...

IncompatibleClassChangeError 276

...no description...

IndexOutOfBoundsException 277

...no description...

InstantiationException 278

...no description...

Integer 279

...no description...

InternalError 286

...no description...

InterruptedException 287

...no description...

InvocationTargetException 288

...no description...

Long 289

...no description...

Math 297

...no description...

NegativeArraySizeException 307

...no description...

NullPointerException 308

...no description...

Number 309

...no description...

NumberFormatException 311

...no description...

Object 311

...no description...

OutOfMemoryError 314

...no description...

RuntimeException 315

...no description...

Short 316

...no description...

StackOverflowError 321

...no description...

StackTraceElement 322

...no description...

StrictMath 324

...no description...

String 334

...no description...

StringBuilder 346

...no description...

StringIndexOutOfBoundsException 353

...no description...

System 354

...no description...

Thread 356

...no description...

Throwable 359

...no description...

UnsatisfiedLinkError 362

...no description...

UnsupportedOperationException 363

...no description...

VirtualMachineError 364

...no description...

Void 365

...no description...

B.1 Interfaces

B.1.1 INTERFACE Appendable

DECLARATION

```
@SCJAllowed
public interface Appendable
```

METHODS

```
@Allocate({javax.safetycritical.annotate.Allocate.Area.THIS})
@SCJAllowed
public Appendable  append( CharSequence csq )
```

```
@Allocate({javax.safetycritical.annotate.Allocate.Area.THIS})
@SCJAllowed
public Appendable  append( CharSequence csq , int start , int end )
```

```
@Allocate({javax.safetycritical.annotate.Allocate.Area.THIS})
@SCJAllowed
public Appendable  append( char c )
```

B.1.2 INTERFACE CharSequence

DECLARATION

```
@SCJAllowed
public interface CharSequence
```

METHODS

```
@BlockFree
@SCJAllowed
public char  charAt( int index )
```

Implementations of this method must not allocate memory and must not allow ”this” to escape the local variables.

```
@BlockFree
@SCJAllowed
public int  length( )
```

Implementations of this method must not allocate memory and must not allow ”this” to escape the local variables.

```
@Allocate({javax.safetycritical.annotate.Allocate.Area.CURRENT})
@BlockFree
@SCJAllowed
public 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.

```
@Allocate({javax.safetycritical.annotate.Allocate.Area.CURRENT})
@BlockFree
@SCJAllowed
public String  toString( )
```

Implementations of this method may allocate a String object in the scope of the caller to hold the result of this method.

This method shall not allow ”this” to escape the local variables.

B.1.3 INTERFACE **Cloneable**

DECLARATION

```
@SCJAllowed
public interface Cloneable
```

Author
jjh

B.1.4 INTERFACE **Comparable**

DECLARATION

```
@SCJAllowed
public interface Comparable
```

METHODS

```
@BlockFree
@SCJAllowed
public int  compareTo( Object o )
```

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 **Deprecated**

DECLARATION

```
@SCJAllowed
@Documented
@Retention(java.lang.annotation.RetentionPolicy.RUNTIME)
public interface Deprecated
  implements java.lang.annotation.Annotation
```

B.1.6 INTERFACE **Override**

DECLARATION

```
@Documented
@Retention(java.lang.annotation.RetentionPolicy.SOURCE)
@SCJAllowed
@Target({java.lang.annotation.ElementType.METHOD})
public interface Override
  implements java.lang.annotation.Annotation
```

B.1.7 INTERFACE **Runnable**

DECLARATION

```
@SCJAllowed
public interface Runnable
```

METHODS

```
@Allocate({javax.safetycritical.annotate.Allocate.Area.CURRENT, javax.safetycritical.annotate.Allocate.A
javax.safetycritical.annotate.Allocate.Area.MISSION, javax.safetycritical.annotate.Allocate.Area.THIS,
javax.safetycritical.annotate.Allocate.Area.SCOPED})
@SCJAllowed
public void run( )
```

The implementation of this method may, in general, perform allocations in immortal memory.

B.1.8 INTERFACE **SuppressWarnings**

DECLARATION

```
@Retention(java.lang.annotation.RetentionPolicy.SOURCE)
@SCJAllowed
@Target({java.lang.annotation.ElementType.TYPE,
java.lang.annotation.ElementType.FIELD,
java.lang.annotation.ElementType.METHOD,
java.lang.annotation.ElementType.PARAMETER,
java.lang.annotation.ElementType.CONSTRUCTOR,
java.lang.annotation.ElementType.LOCAL_VARIABLE})
public interface SuppressWarnings
  implements java.lang.annotation.Annotation
```

B.1.9 INTERFACE **Thread.UncaughtExceptionHandler**

DECLARATION

```
@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
Allocates no memory. Does not allow implicit argument this, or explit arguments t and e to escape local variables.

B.2 Classes

B.2.1 CLASS ArithmeticException

DECLARATION

```
@SCJAllowed
public class ArithmeticException
  implements java.io.Serializable
  extends java.lang.RuntimeException
```

CONSTRUCTORS

@Allocate({javax.safetycritical.annotate.Allocate.Area.CURRENT})
@BlockFree
@SCJAllowed
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).

@Allocate({javax.safetycritical.annotate.Allocate.Area.CURRENT})
@BlockFree
@MemoryAreaEncloses(inner = {”this”}, outer = {”msg”})
@SCJAllowed
public **ArithmeticException**(String msg)

Shall not copy ”this” to any instance or static field. The scope containing the msg argument must enclose the scope containing ”this”. Otherwise, an IllegalAssignmentError will be thrown.

Allocates an application- and implementation-dependent amount of memory in the current scope (to represent stack backtrace).

B.2.2 CLASS **ArrayIndexOutOfBoundsException**

DECLARATION

```
@SCJAllowed
public class ArrayIndexOutOfBoundsException
  implements java.io.Serializable
  extends java.lang.IndexOutOfBoundsException
```

CONSTRUCTORS

```
@Allocate({javax.safetycritical.annotate.Allocate.Area.CURRENT})
@BlockFree
@SCJAllowed
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).

```
@Allocate({javax.safetycritical.annotate.Allocate.Area.CURRENT})
@BlockFree
@SCJAllowed
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).

```
@Allocate({javax.safetycritical.annotate.Allocate.Area.CURRENT})
@BlockFree
@MemoryAreaEncloses(inner = {"this"}, outer = {"msg"})
@SCJAllowed
public ArrayIndexOutOfBoundsException( String msg )
```

Shall not copy "this" to any instance or static field. The scope containing the msg argument must enclose the scope containing "this". Otherwise, an IllegalAssignmentError will be thrown.

Allocates an application- and implementation-dependent amount of memory in the current scope (to represent stack backtrace).

B.2.3 CLASS ArrayStoreException

DECLARATION

```
@SCJAllowed
public class ArrayStoreException
  implements java.io.Serializable
  extends java.lang.RuntimeException
```

CONSTRUCTORS

```
@Allocate({javax.safetycritical.annotate.Allocate.Area.CURRENT})
@BlockFree
@SCJAllowed
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).

```
@Allocate({javax.safetycritical.annotate.Allocate.Area.CURRENT})
@BlockFree
@MemoryAreaEncloses(inner = {"this"}, outer = {"msg"})
@SCJAllowed
public ArrayStoreException( String msg )
```

Shall not copy ”this” to any instance or static field. The scope containing the msg argument must enclose the scope containing ”this”. Otherwise, an IllegalAssignmentError will be thrown.

Allocates an application- and implementation-dependent amount of memory in the current scope (to represent stack backtrace).

B.2.4 CLASS **AssertionError**

DECLARATION

```
@SCJAllowed
public class AssertionError
  implements java.io.Serializable
  extends java.lang.Error
```

CONSTRUCTORS

```
@Allocate({javax.safetycritical.annotate.Allocate.Area.CURRENT})
@BlockFree
@SCJAllowed
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).

```
@Allocate({javax.safetycritical.annotate.Allocate.Area.CURRENT})
@BlockFree
@SCJAllowed
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).

@Allocate({javax.safetycritical.annotate.Allocate.Area.CURRENT})
@BlockFree
@SCJAllowed
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).

@Allocate({javax.safetycritical.annotate.Allocate.Area.CURRENT})
@BlockFree
@SCJAllowed
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).

@Allocate({javax.safetycritical.annotate.Allocate.Area.CURRENT})
@BlockFree
@SCJAllowed
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).

@Allocate({javax.safetycritical.annotate.Allocate.Area.CURRENT})
@BlockFree
@SCJAllowed
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).

```
@Allocate({javax.safetycritical.annotate.Allocate.Area.CURRENT})
@BlockFree
@SCJAllowed
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).

```
@Allocate({javax.safetycritical.annotate.Allocate.Area.CURRENT})
@BlockFree
@MemoryAreaEncloses(inner = {”this”}, outer = {”o”})
@SCJAllowed
public AssertionError( Object o )
```

Shall not copy ”this” to any instance or static field.

Allocates an application- and implementation-dependent amount of memory in the current scope (to represent stack backtrace).

B.2.5 CLASS **BigDecimal**

DECLARATION

```
@SCJAllowed
public class BigDecimal
  implements java.lang.Comparable
  extends java.lang.Number
```

FIELDS

```
 @SCJAllowed
public static final BigDecimal ONE
```

```
 @SCJAllowed
public static final int ROUND_CELING
```

@SCJAllowed
public static final int ROUND_DOWN

@SCJAllowed
public static final int ROUND_FLOOR

@SCJAllowed
public static final int ROUND_HALF_DOWN

@SCJAllowed
public static final int ROUND_HALF_EVEN

@SCJAllowed
public static final int ROUND_HALF_UP

@SCJAllowed
public static final int ROUND_UNNECESSARY

@SCJAllowed
public static final int ROUND_UP

@SCJAllowed
public static final BigDecimal TEN

@SCJAllowed
public static final BigDecimal ZERO

CONSTRUCTORS

@Allocate({javax.safetycritical.annotate.Allocate.Area.THIS})
@BlockFree

@SCJAllowed
public **BigDecimal**(BigInteger val)

Does not allow ”this” or ”val” to escape local variables.

@Allocate({javax.safetycritical.annotate.Allocate.Area.THIS})
@BlockFree
@SCJAllowed
public **BigDecimal**(BigInteger val , int scale)

Does not allow ”this” or ”val” to escape local variables.

@Allocate({javax.safetycritical.annotate.Allocate.Area.THIS})
@BlockFree
@SCJAllowed
public **BigDecimal**(char []in)

Does not allow ”this” or ”in” to escape local variables.

@Allocate({javax.safetycritical.annotate.Allocate.Area.THIS})
@BlockFree
@SCJAllowed
public **BigDecimal**(char []in , int offset , int len)

Does not allow ”this” or ”in” to escape local variables.

@Allocate({javax.safetycritical.annotate.Allocate.Area.THIS})
@BlockFree
@SCJAllowed
public **BigDecimal**(double val)

Does not allow ”this” to escape local variables.

@Allocate({javax.safetycritical.annotate.Allocate.Area.THIS})
@BlockFree
@SCJAllowed
public **BigDecimal**(int val)

Does not allow ”this” to escape local variables.

@Allocate({javax.safetycritical.annotate.Allocate.Area.THIS})
@BlockFree
@SCJAllowed
public **BigDecimal**(long val)

Does not allow ”this” to escape local variables.

@Allocate({javax.safetycritical.annotate.Allocate.Area.THIS})
@BlockFree
@SCJAllowed
public **BigDecimal**(String val)

Does not allow ”this” or ”val” to escape local variables.

METHODS

@Allocate({javax.safetycritical.annotate.Allocate.Area.CURRENT})
@BlockFree
@SCJAllowed
public BigDecimal **abs**( )

Does not allow ”this” to escape local variables.

@Allocate({javax.safetycritical.annotate.Allocate.Area.CURRENT})
@BlockFree
@SCJAllowed
public BigDecimal **add**(BigDecimal val)

Does not allow ”this” to escape local variables.

@BlockFree
@SCJAllowed
public byte **byteValueExact**( )

Does not allow ”this” to escape local variables.

@BlockFree
@SCJAllowed
public int **compareTo**(BigDecimal val)

Does not allow ”this” or ”val” to escape local variables.

@Allocate({javax.safetycritical.annotate.Allocate.Area.CURRENT})
@BlockFree
@SCJAllowed
public BigDecimal **divide**(BigDecimal val)

Does not allow ”this” to escape local variables.

@Allocate({javax.safetycritical.annotate.Allocate.Area.CURRENT})
@BlockFree
@SCJAllowed
public BigDecimal [] **divideAndRemainder**(BigDecimal val)

Does not allow ”this” to escape local variables.

@Allocate({javax.safetycritical.annotate.Allocate.Area.CURRENT})
@BlockFree
@SCJAllowed
public BigDecimal [] **divideToIntegralValue**(BigDecimal val)

Does not allow ”this” to escape local variables.

```
@BlockFree
@SCJAllowed
public double doubleValue( BigDecimal val )
```

Allocates no memory. Does not allow ”this” to escape local variables.

```
@BlockFree
@SCJAllowed
public boolean equals( Object x )
```

Does not allow ”this” or ”x” to escape local variables.

```
@BlockFree
@SCJAllowed
public float floatValue( BigDecimal val )
```

Allocates no memory. Does not allow ”this” to escape local variables.

```
@BlockFree
@SCJAllowed
public int hashCode( )
```

Allocates no memory. Does not allow ”this” to escape local variables.

```
@BlockFree
@SCJAllowed
public int intValue( )
```

Allocates no memory. Does not allow ”this” to escape local variables.

@BlockFree
@SCJAllowed
public int **intValueExact**( )

Allocates no memory. Does not allow ”this” to escape local variables.

@BlockFree
@SCJAllowed
public long **longValue**( )

Allocates no memory. Does not allow ”this” to escape local variables.

@BlockFree
@SCJAllowed
public int **longValueExact**( )

Allocates no memory. Does not allow ”this” to escape local variables.

@Allocate({javax.safetycritical.annotate.Allocate.Area.CURRENT})
@BlockFree
@SCJAllowed
public BigDecimal **max**(BigDecimal val)

Does not allow ”this” or ”max” to escape local variables.

@Allocate({javax.safetycritical.annotate.Allocate.Area.CURRENT})
@BlockFree
@SCJAllowed
public BigDecimal **min**(BigDecimal val)

Does not allow ”this” or ”val” to escape local variables.

@Allocate({javax.safetycritical.annotate.Allocate.Area.CURRENT})
@BlockFree
@SCJAllowed
public BigDecimal **movePointLeft**(int n)

Does not allow ”this” to escape local variables.

@Allocate({javax.safetycritical.annotate.Allocate.Area.CURRENT})
@BlockFree
@SCJAllowed
public BigDecimal **movePointRight**(int n)

Does not allow ”this” to escape local variables.

@Allocate({javax.safetycritical.annotate.Allocate.Area.CURRENT})
@BlockFree
@SCJAllowed
public BigDecimal **multiply**(BigDecimal val)

Does not allow ”this” or ”val” to escape local variables.

@Allocate({javax.safetycritical.annotate.Allocate.Area.CURRENT})
@BlockFree
@SCJAllowed
public BigDecimal **negate**( )

Does not allow ”this” to escape local variables.

@Allocate({javax.safetycritical.annotate.Allocate.Area.CURRENT})
@BlockFree
@SCJAllowed
public BigDecimal **plus**( )

Does not allow ”this” to escape local variables.

@Allocate({javax.safetycritical.annotate.Allocate.Area.CURRENT})
@BlockFree
@SCJAllowed
public BigDecimal **pow**(int exponent)

Does not allow ”this” to escape local variables.

@BlockFree
@SCJAllowed
public int **precision**( )

Allocates no memory. Does not allow ”this” to escape local variables.

@Allocate({javax.safetycritical.annotate.Allocate.Area.CURRENT})
@BlockFree
@SCJAllowed
public BigDecimal **remainder**(BigDecimal val)

Does not allow ”this” or ”val” to escape local variables.

@BlockFree
@SCJAllowed
public int **scale**( )

Allocates no memory. Does not allow ”this” to escape local variables.

@Allocate({javax.safetycritical.annotate.Allocate.Area.CURRENT})
@BlockFree
@SCJAllowed
public BigDecimal **scaleByPowerOfTen**(int n)

Does not allow ”this” to escape local variables.

@Allocate({javax.safetycritical.annotate.Allocate.Area.CURRENT})
@BlockFree
@SCJAllowed
public BigDecimal **setScale**(int newScale , int roundingMode)

Does not allow ”this” to escape local variables.

@Allocate({javax.safetycritical.annotate.Allocate.Area.CURRENT})
@BlockFree
@SCJAllowed
public BigDecimal **setScale**(int newScale)

Does not allow ”this” to escape local variables.

@BlockFree
@SCJAllowed
public short **shortValueExact**( )

Allocates no memory. Does not allow ”this” to escape local variables.

@BlockFree
@SCJAllowed
public int **signum**( )

Allocates no memory. Does not allow ”this” to escape local variables.

@Allocate({javax.safetycritical.annotate.Allocate.Area.CURRENT})
@BlockFree
@SCJAllowed
public BigDecimal **stripTrailingZeros**( )

Does not allow ”this” to escape local variables.

@Allocate({javax.safetycritical.annotate.Allocate.Area.CURRENT})
@BlockFree
@SCJAllowed
public BigDecimal **subtract**(BigDecimal val)

Does not allow ”this” or ”val” to escape local variables.

@Allocate({javax.safetycritical.annotate.Allocate.Area.CURRENT})
@BlockFree
@SCJAllowed
public BigInteger **toBigInteger**( )

Does not allow ”this” or ”val” to escape local variables.

@Allocate({javax.safetycritical.annotate.Allocate.Area.CURRENT})
@BlockFree
@SCJAllowed
public String **toEngineeringString**( )

Does not allow ”this” or ”val” to escape local variables.

@Allocate({javax.safetycritical.annotate.Allocate.Area.CURRENT})
@BlockFree
@SCJAllowed
public String **toPlainString**( )

Does not allow ”this” or ”val” to escape local variables.

@Allocate({javax.safetycritical.annotate.Allocate.Area.CURRENT})
@BlockFree
@SCJAllowed
public String **toString**( )

Does not allow ”this” to escape local variables.

@Allocate({javax.safetycritical.annotate.Allocate.Area.CURRENT})
@BlockFree
@SCJAllowed
public BigDecimal **ulp**( )

Does not allow ”this” to escape local variables.

@Allocate({javax.safetycritical.annotate.Allocate.Area.CURRENT})
@BlockFree
@SCJAllowed
public BigInteger **unscaledValue**( )

Does not allow ”this” to escape local variables.

@Allocate({javax.safetycritical.annotate.Allocate.Area.CURRENT})
@BlockFree
@SCJAllowed
public static BigDecimal **valueOf**(long unscaledVal , int scale)

@Allocate({javax.safetycritical.annotate.Allocate.Area.CURRENT})
@BlockFree
@SCJAllowed
public static BigDecimal **valueOf**(long val)

@Allocate({javax.safetycritical.annotate.Allocate.Area.CURRENT})
@BlockFree
@SCJAllowed
public static BigDecimal **valueOf**(double val)

B.2.6 CLASS **BigInteger**

DECLARATION

```
@SCJAllowed
public class BigInteger
  implements java.lang.Comparable
  extends java.lang.Number
```

FIELDS

```
 @SCJAllowed
public static final BigInteger ONE
```

```
 @SCJAllowed
public static final BigInteger TEN
```

```
 @SCJAllowed
public static final BigInteger ZERO
```

CONSTRUCTORS

```
@Allocate({javax.safetycritical.annotate.Allocate.Area.THIS})
@BlockFree
@SCJAllowed
public BigInteger( byte []val )
```

Does not allow ”this” to escape local variables.

```
@Allocate({javax.safetycritical.annotate.Allocate.Area.THIS})
@BlockFree
@SCJAllowed
public BigInteger( int signum , byte []magnitude )
```

Does not allow ”this” to escape local variables.

@Allocate({javax.safetycritical.annotate.Allocate.Area.THIS})
@BlockFree
@SCJAllowed
public **BigInteger**(String val)

Does not allow ”this” to escape local variables.

@Allocate({javax.safetycritical.annotate.Allocate.Area.THIS})
@BlockFree
@SCJAllowed
public **BigInteger**(String val , int radix)

Does not allow ”this” to escape local variables.

METHODS

@Allocate({javax.safetycritical.annotate.Allocate.Area.CURRENT})
@BlockFree
@SCJAllowed
public BigInteger **abs**( )

Does not allow ”this” to escape local variables.

@Allocate({javax.safetycritical.annotate.Allocate.Area.CURRENT})
@BlockFree
@SCJAllowed
public BigInteger **add**(BigInteger val)

Does not allow ”this” to escape local variables.

@Allocate({javax.safetycritical.annotate.Allocate.Area.CURRENT})
@BlockFree
@SCJAllowed
public BigInteger **andNot**(BigInteger val)

Does not allow ”this” to escape local variables.

```
@BlockFree
@SCJAllowed
public int bitCount( )
```

Allocates no memory. Does not allow ”this” to escape local variables.

```
@BlockFree
@SCJAllowed
public int bitLength( )
```

Allocates no memory. Does not allow ”this” to escape local variables.

```
@Allocate({javax.safetycritical.annotate.Allocate.Area.CURRENT})
@BlockFree
@SCJAllowed
public BigInteger clearBit( int n )
```

Does not allow ”this” to escape local variables.

```
@BlockFree
@SCJAllowed
public int compareTo( BigInteger val )
```

Does not allow ”this” to escape local variables.

```
@Allocate({javax.safetycritical.annotate.Allocate.Area.CURRENT})
@BlockFree
@SCJAllowed
public BigInteger divide( BigInteger val )
```

Does not allow ”this” to escape local variables.

@Allocate({javax.safetycritical.annotate.Allocate.Area.CURRENT})
@BlockFree
@SCJAllowed
public BigInteger [] **divideAndRemainder**(BigInteger val)

Does not allow ”this” to escape local variables.

@BlockFree
@SCJAllowed
public double **doubleValue**(BigInteger val)

Allocates no memory. Does not allow ”this” to escape local variables.

@BlockFree
@SCJAllowed
public boolean **equals**(Object x)

Does not allow ”this” or ”x” to escape local variables.

@Allocate({javax.safetycritical.annotate.Allocate.Area.CURRENT})
@BlockFree
@SCJAllowed
public BigInteger **flipBit**(int n)

Does not allow ”this” to escape local variables.

@BlockFree
@SCJAllowed
public float **floatValue**(BigInteger val)

Allocates no memory. Does not allow ”this” to escape local variables.

```
@Allocate({javax.safetycritical.annotate.Allocate.Area.CURRENT})
@BlockFree
@SCJAllowed
public BigInteger  gcd( BigInteger val )
```

Does not allow ”this” to escape local variables.

```
@BlockFree
@SCJAllowed
public int  getLowestSetBit( )
```

Allocates no memory. Does not allow ”this” to escape local variables.

```
@BlockFree
@SCJAllowed
public int  hashCode( )
```

Allocates no memory. Does not allow ”this” to escape local variables.

```
@BlockFree
@SCJAllowed
public int  intValue( )
```

Allocates no memory. Does not allow ”this” to escape local variables.

```
@BlockFree
@SCJAllowed
public boolean  isProbablePrime( int certainty )
```

Allocates no memory. Does not allow ”this” to escape local variables.

```
@BlockFree
@SCJAllowed
public long  longValue( )
```

Allocates no memory. Does not allow ”this” to escape local variables.

@Allocate({javax.safetycritical.annotate.Allocate.Area.CURRENT})
@BlockFree
@SCJAllowed
public BigInteger **max**(BigInteger val)

Does not allow ”this” or ”max” to escape local variables.

@Allocate({javax.safetycritical.annotate.Allocate.Area.CURRENT})
@BlockFree
@SCJAllowed
public BigInteger **min**(BigInteger val)

Does not allow ”this” or ”val” to escape local variables.

@Allocate({javax.safetycritical.annotate.Allocate.Area.CURRENT})
@BlockFree
@SCJAllowed
public BigInteger **mod**(BigInteger val)

Does not allow ”this” or ”val” to escape local variables.

@Allocate({javax.safetycritical.annotate.Allocate.Area.CURRENT})
@BlockFree
@SCJAllowed
public BigInteger **modInverse**(BigInteger val)

Does not allow ”this” or ”val” to escape local variables.

@Allocate({javax.safetycritical.annotate.Allocate.Area.CURRENT})
@BlockFree
@SCJAllowed
public BigInteger **modPow**(BigInteger exponent , BigInteger m)

Does not allow ”this”, ”exponent”, or ”m” to escape local variables.

```
@Allocate({javax.safetycritical.annotate.Allocate.Area.CURRENT})
@BlockFree
@SCJAllowed
public BigInteger multiply( BigInteger val )
```

Does not allow ”this” or val to escape local variables.

```
@Allocate({javax.safetycritical.annotate.Allocate.Area.CURRENT})
@BlockFree
@SCJAllowed
public BigInteger negate( )
```

Does not allow ”this” to escape local variables.

```
@Allocate({javax.safetycritical.annotate.Allocate.Area.CURRENT})
@BlockFree
@SCJAllowed
public BigInteger nextProbablePrime( )
```

Does not allow ”this” to escape local variables.

```
@Allocate({javax.safetycritical.annotate.Allocate.Area.CURRENT})
@BlockFree
@SCJAllowed
public BigInteger not( )
```

Does not allow ”this” to escape local variables.

```
@Allocate({javax.safetycritical.annotate.Allocate.Area.CURRENT})
@BlockFree
@SCJAllowed
public BigInteger or( BigInteger val )
```

Does not allow ”this” or ”val” to escape local variables.

@Allocate({javax.safetycritical.annotate.Allocate.Area.CURRENT})
@BlockFree
@SCJAllowed
public BigInteger **pow**(int exponent)

Does not allow ”this” to escape local variables.

@Allocate({javax.safetycritical.annotate.Allocate.Area.CURRENT})
@BlockFree
@SCJAllowed
public BigInteger **remainder**(BigInteger val)

Does not allow ”this” or ”val” to escape local variables.

@Allocate({javax.safetycritical.annotate.Allocate.Area.CURRENT})
@BlockFree
@SCJAllowed
public BigInteger **setBit**(int n)

Does not allow ”this” to escape local variables.

@Allocate({javax.safetycritical.annotate.Allocate.Area.CURRENT})
@BlockFree
@SCJAllowed
public BigInteger **shiftLeft**(int n)

Does not allow ”this” to escape local variables.

@Allocate({javax.safetycritical.annotate.Allocate.Area.CURRENT})
@BlockFree
@SCJAllowed
public BigInteger **shiftRight**(int n)

Does not allow ”this” to escape local variables.

```
@BlockFree
@SCJAllowed
public int signum( )
```

Allocates no memory. Does not allow ”this” to escape local variables.

```
@Allocate({javax.safetycritical.annotate.Allocate.Area.CURRENT})
@BlockFree
@SCJAllowed
public BigInteger subtract( BigInteger val )
```

Does not allow ”this” or ”val” to escape local variables.

```
@BlockFree
@SCJAllowed
public boolean testBit( int n )
```

Allocates no memory. Does not allow ”this” to escape local variables.

```
@Allocate({javax.safetycritical.annotate.Allocate.Area.CURRENT})
@BlockFree
@SCJAllowed
public BigInteger toByteArray( )
```

Does not allow ”this” to escape local variables.

```
@Allocate({javax.safetycritical.annotate.Allocate.Area.CURRENT})
@BlockFree
@SCJAllowed
public String toString( int radix )
```

Does not allow ”this” to escape local variables.

```
@Allocate({javax.safetycritical.annotate.Allocate.Area.CURRENT})
@BlockFree
@SCJAllowed
public String  toString( )
```

Does not allow ”this” to escape local variables.

```
@Allocate({javax.safetycritical.annotate.Allocate.Area.CURRENT})
@BlockFree
@SCJAllowed
public static BigInteger  valueOf( long val )
```

```
@Allocate({javax.safetycritical.annotate.Allocate.Area.CURRENT})
@BlockFree
@SCJAllowed
public BigInteger  xor( BigInteger val )
```

Does not allow ”this” or ”val” to escape local variables.

B.2.7 CLASS **Boolean**

DECLARATION

```
@SCJAllowed
public class Boolean
  implements java.lang.Comparable, java.io.Serializable
  extends java.lang.Object
```

FIELDS

```
 @SCJAllowed
public static final Boolean FALSE
```

@SCJAllowed
public static final Boolean TRUE

@SCJAllowed
public static final Class TYPE

CONSTRUCTORS

@BlockFree
@SCJAllowed
public **Boolean**(boolean v)

Allocates no memory. Does not allow ”this” to escape local variables.

@BlockFree
@SCJAllowed
public **Boolean**(String str)

Allocates no memory. Does not allow ”this” or ”str” argument to escape local variables.

METHODS

@BlockFree
@SCJAllowed
public boolean **booleanValue**( )

Allocates no memory. Does not allow ”this” to escape local variables.

@BlockFree
@SCJAllowed
public int **compareTo**(Boolean b)

Allocates no memory. Does not allow ”this” or argument ”b” to escape local variables.

@BlockFree
@Override
@SCJAllowed
public boolean **equals**(Object obj)

Allocates no memory. Does not allow ”this” or argument ”obj” to escape local variables.

@BlockFree
@SCJAllowed
public static boolean **getBoolean**(String str)

Allocates no memory. Does not allow argument ”str” to escape local variables.

@BlockFree
@Override
@SCJAllowed
public int **hashCode**( )

Allocates no memory. Does not allow ”this” to escape local variables.

@BlockFree
@SCJAllowed
public static boolean **parseBoolean**(String str)

Allocates no memory. Does not allow argument ”str” to escape local variables.

@BlockFree
@SCJAllowed
public static 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.

```
@Allocate({javax.safetycritical.annotate.Allocate.Area.CURRENT})
@BlockFree
@Override
@SCJAllowed
public 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()).

```
@BlockFree
@SCJAllowed
public static 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.

```
@BlockFree
@SCJAllowed
public static 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.2.8 CLASS Byte

DECLARATION

```
@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 Class TYPE

CONSTRUCTORS

@BlockFree
@SCJAllowed
public **Byte**(byte val)

Allocates no memory. Does not allow ”this” to escape local variables.

@BlockFree
@SCJAllowed
public **Byte**(String str)

Allocates no memory. Does not allow ”this” or ”str” argument to escape local variables.

METHODS

@BlockFree
@SCJAllowed
public byte **byteValue**( )

Allocates no memory. Does not allow ”this” to escape local variables.

@BlockFree
@SCJAllowed
public int **compareTo**(Byte other)

Allocates no memory. Does not allow ”this” or ”other” argument to escape local variables.

@BlockFree
@SCJAllowed
public static Byte **decode**(String str)

Does not allow ”str” argument to escape local variables. Allocates a Byte result object in the caller’s scope.

@BlockFree
@SCJAllowed
public double **doubleValue**( )

Allocates no memory. Does not allow ”this” to escape local variables.

@BlockFree
@SCJAllowed
public boolean **equals**(Object obj)

Allocates no memory. Does not allow ”this” or ”obj” argument to escape local variables.

@BlockFree
@SCJAllowed
public float **floatValue**( )

Allocates no memory. Does not allow ”this” to escape local variables.

@BlockFree
@SCJAllowed
public int **hashCode**( )

Allocates no memory. Does not allow ”this” to escape local variables.

@BlockFree
@SCJAllowed
public int **intValue**( )

Allocates no memory. Does not allow ”this” to escape local variables.

@BlockFree
@SCJAllowed
public long **longValue**( )

Allocates no memory. Does not allow ”this” to escape local variables.

@BlockFree
@SCJAllowed
public static byte **parseByte**(String str , int base)

Allocates no memory. Does not allow ”str” argument to escape local variables.

@BlockFree
@SCJAllowed
public static byte **parseByte**(String str)

Allocates no memory. Does not allow ”str” argument to escape local variables.

@BlockFree
@SCJAllowed
public short **shortValue**( )

Allocates no memory. Does not allow ”this” to escape local variables.

@Allocate({javax.safetycritical.annotate.Allocate.Area.CURRENT})
@BlockFree
@SCJAllowed
public static String **toString**(byte v)

Allocates a String and associated internal ”structure” (e.g. char[]) in caller’s scope.

@Allocate({javax.safetycritical.annotate.Allocate.Area.CURRENT})
@BlockFree
@SCJAllowed
public 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()).

@Allocate({javax.safetycritical.annotate.Allocate.Area.CURRENT})
@BlockFree
@SCJAllowed
public static Byte **valueOf**(String str , int base)

Does not allow ”str” argument to escape local variables. Allocates one Byte object in the caller’s scope.

@Allocate({javax.safetycritical.annotate.Allocate.Area.CURRENT})
@BlockFree
@SCJAllowed
public static Byte **valueOf**(byte val)

Allocates one Byte object in the caller's scope.

```
@Allocate({javax.safetycritical.annotate.Allocate.Area.CURRENT})
@BlockFree
@SCJAllowed
public static Byte  valueOf( String str )
```

Does not allow "str" argument to escape local variables. Allocates one Byte object in the caller's scope.

B.2.9 CLASS Character

DECLARATION

```
@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 Class TYPE

@SCJAllowed
public static final byte UNASSIGNED

@SCJAllowed
public static final byte UPPERCASE_LETTER

CONSTRUCTORS

@BlockFree
@SCJAllowed
public **Character**(char v)

Allocates no memory. Does not allow ”this” to escape local variables.

METHODS

@BlockFree
@SCJAllowed
public char **charValue**( )

Allocates no memory. Does not allow ”this” to escape local variables.

@BlockFree
@SCJAllowed
public int **compareTo**(Character another_character)

Allocates no memory. Does not allow ”this” or ”another_character” argument to escape local variables.

@BlockFree
@SCJAllowed
public static int **digit**(char ch , int radix)

Allocates no memory.

@BlockFree
@SCJAllowed
public boolean **equals**(Object obj)

Allocates no memory. Does not allow ”this” or ”obj” argument to escape local variables.

@BlockFree
@SCJAllowed
public static int **getType**(char ch)

Allocates no memory.

@BlockFree
@SCJAllowed
public int **hashCode**( )

@BlockFree
@SCJAllowed
public static boolean **isLetter**(char ch)

Allocates no memory.

@BlockFree
@SCJAllowed
public static boolean **isLetterOrDigit**(char ch)

Allocates no memory.

@BlockFree
@SCJAllowed
public static boolean **isLowerCase**(char ch)

Allocates no memory.

@BlockFree
@SCJAllowed
public static boolean **isSpaceChar**(char ch)

Allocates no memory.

@BlockFree
@SCJAllowed
public static boolean **isUpperCase**(char ch)

Allocates no memory.

@BlockFree
@SCJAllowed
public static boolean **isWhitespace**(char ch)

Allocates no memory.

@BlockFree
@SCJAllowed
public static char **toLowerCase**(char ch)

Allocates no memory.

@Allocate({javax.safetycritical.annotate.Allocate.Area.CURRENT})
@BlockFree
@SCJAllowed
public static String **toString**(char c)

Allocates a String and associated internal ”structure” (e.g. char[]) in caller’s scope.

@Allocate({javax.safetycritical.annotate.Allocate.Area.CURRENT})
@BlockFree
@SCJAllowed
public 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()).

@BlockFree
@SCJAllowed
public static char **toUpperCase**(char ch)

Allocates no memory.

@Allocate({javax.safetycritical.annotate.Allocate.Area.CURRENT})
@BlockFree
@SCJAllowed
public static Character **valueOf**(char c)

Allocates a Character object in caller’s scope.

B.2.10 CLASS Class

DECLARATION

```
@SCJAllowed
public final class Class
  implements java.io.Serializable
  extends java.lang.Object
```

METHODS

@BlockFree
@SCJAllowed
public boolean **desiredAssertionStatus**( )

Allocates no memory. Does not allow ”this” to escape local variables.

```
@BlockFree
@SCJAllowed
public 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.

```
@BlockFree
@SCJAllowed
public Class getDeclaringClass( )
```

Allocates no memory. Returns a reference to a previously existing Class, which resides in the scope of its ClassLoader.

```
@Allocate({javax.safetycritical.annotate.Allocate.Area.CURRENT})
@BlockFree
@MemoryAreaEncloses(inner = {”@result”}, outer = {”this.getClass().getClassLoader()”})
@SCJAllowed
public Object [] 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.

```
@BlockFree
@SCJAllowed
public 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.

@BlockFree
@SCJAllowed
public Class **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.

@BlockFree
@SCJAllowed
public boolean **isAnnotation**( )

Allocates no memory. Does not allow ”this” to escape local variables.

@BlockFree
@SCJAllowed
public boolean **isArray**( )

Allocates no memory. Does not allow ”this” to escape local variables.

@BlockFree
@SCJAllowed
public boolean **isAssignableFrom**(Class c)

Allocates no memory. Does not allow ”this” or argument ”c” to escape local variables.

@BlockFree
@SCJAllowed
public boolean **isEnum**( )

Allocates no memory. Does not allow ”this” to escape local variables.

```
@BlockFree
@SCJAllowed
public boolean isInstance( Object o )
```

Allocates no memory. Does not allow ”this” to escape local variables.

```
@BlockFree
@SCJAllowed
public boolean isInterface( )
```

Allocates no memory. Does not allow ”this” to escape local variables.

```
@BlockFree
@SCJAllowed
public boolean isPrimitive( )
```

Allocates no memory. Does not allow ”this” to escape local variables.

```
@Allocate({javax.safetycritical.annotate.Allocate.Area.CURRENT})
@BlockFree
@SCJAllowed
public String toString( )
```

Does not allow ”this” to escape local variables. Allocates a String and associated internal ”structure” (e.g. char[]) in caller’s scope. (Note: this semantics is desired for consistency with overridden implementation of Object.toString()).

B.2.11 CLASS **ClassCastException**

DECLARATION

```
@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).

@Allocate({javax.safetycritical.annotate.Allocate.Area.CURRENT})
@BlockFree
@MemoryAreaEncloses(inner = {”this”}, outer = {”msg”})
@SCJAllowed
public **ClassCastException**(String msg)

Shall not copy ”this” to any instance or static field. The scope containing the msg argument must enclose the scope containing ”this”. Otherwise, an IllegalAssignmentError will be thrown.

Allocates an application- and implementation-dependent amount of memory in the current scope (to represent stack backtrace).

B.2.12 CLASS **ClassNotFoundException**

DECLARATION

```
@SCJAllowed
public class ClassNotFoundException
  implements java.io.Serializable
  extends java.lang.Exception
```

CONSTRUCTORS

@Allocate({javax.safetycritical.annotate.Allocate.Area.CURRENT})
@BlockFree
@SCJAllowed
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).

```
@Allocate({javax.safetycritical.annotate.Allocate.Area.CURRENT})
@BlockFree
@MemoryAreaEncloses(inner = {”this”}, outer = {”msg”})
@SCJAllowed
public ClassNotFoundException( String msg )
```

Shall not copy ”this” to any instance or static field. The scope containing the msg argument must enclose the scope containing ”this”. Otherwise, an IllegalAssignmentError will be thrown.

Allocates an application- and implementation-dependent amount of memory in the current scope (to represent stack backtrace).

B.2.13 CLASS CloneNotSupportedException

DECLARATION

```
@SCJAllowed
public class CloneNotSupportedException
  implements java.io.Serializable
  extends java.lang.Exception
```

CONSTRUCTORS

```
@Allocate({javax.safetycritical.annotate.Allocate.Area.CURRENT})
@BlockFree
@SCJAllowed
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).

```
@Allocate({javax.safetycritical.annotate.Allocate.Area.CURRENT})
@BlockFree
@MemoryAreaEncloses(inner = {"this"}, outer = {"msg"})
@SCJAllowed
public CloneNotSupportedException( String msg )
```

Shall not copy "this" to any instance or static field. The scope containing the msg argument must enclose the scope containing "this". Otherwise, an IllegalAssignmentError will be thrown.

Allocates an application- and implementation-dependent amount of memory in the current scope (to represent stack backtrace).

B.2.14 CLASS Double

DECLARATION

```
@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 Class TYPE

CONSTRUCTORS

@BlockFree
@SCJAllowed
public **Double**(double val)

Allocates no memory. Does not allow ”this” to escape local variables.

@BlockFree
@SCJAllowed
public **Double**(String str)

Allocates no memory. Does not allow ”this” or ”str” argument to escape local variables.

METHODS

@BlockFree
@SCJAllowed
public byte **byteValue**( )

Allocates no memory. Does not allow ”this” to escape local variables.

@BlockFree
@SCJAllowed
public static int **compare**(double value1 , double value2)

Allocates no memory.

@BlockFree
@SCJAllowed
public int **compareTo**(Double other)

Allocates no memory. Does not allow ”this” or ”other” argument to escape local variables.

@BlockFree
@SCJAllowed
public static long **doubleToLongBits**(double v)

Allocates no memory.

@BlockFree
@SCJAllowed
public static native long **doubleToRawLongBits**(double val)

Allocates no memory.

@BlockFree
@SCJAllowed
public double **doubleValue**( )

Allocates no memory. Does not allow ”this” to escape local variables.

@BlockFree
@SCJAllowed
public boolean **equals**(Object obj)

Allocates no memory. Does not allow ”this” or ”obj” argument to escape local variables.

@BlockFree
@SCJAllowed
public float **floatValue**( )

Allocates no memory. Does not allow ”this” to escape local variables.

@BlockFree
@SCJAllowed
public int **hashCode**( )

Allocates no memory. Does not allow ”this” to escape local variables.

@BlockFree
@SCJAllowed
public int **intValue**( )

Allocates no memory. Does not allow ”this” to escape local variables.

@BlockFree
@SCJAllowed
public static boolean **isInfinite**(double v)

Allocates no memory.

@BlockFree
@SCJAllowed
public boolean **isInfinite**( )

Allocates no memory. Does not allow ”this” to escape local variables.

@BlockFree
@SCJAllowed
public static boolean **isNaN**(double v)

Allocates no memory.

@BlockFree
@SCJAllowed
public boolean **isNaN**( )

Allocates no memory. Does not allow ”this” to escape local variables.

@BlockFree
@SCJAllowed
public static double **longBitsToDouble**(long v)

Allocates no memory.

@BlockFree
@SCJAllowed
public long **longValue**( )

Allocates no memory. Does not allow ”this” to escape local variables.

@BlockFree
@SCJAllowed
public static double **parseDouble**(String s)

Does not allow ”this” to escape local variables.

@BlockFree
@SCJAllowed
public short **shortValue**( )

Allocates no memory. Does not allow ”this” to escape local variables.

@Allocate({javax.safetycritical.annotate.Allocate.Area.CURRENT})
@BlockFree
@SCJAllowed
public static String **toString**(double v)

Allocates a String and associated internal ”structure” (e.g. char[]) in caller’s scope.

@Allocate({javax.safetycritical.annotate.Allocate.Area.CURRENT})
@BlockFree
@SCJAllowed
public 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()).

@Allocate({javax.safetycritical.annotate.Allocate.Area.CURRENT})
@BlockFree
@SCJAllowed
public static Double **valueOf**(String str)

Does not allow ”this” to escape local variables. Allocates a Double in caller’s scope.

```
@Allocate({javax.safetycritical.annotate.Allocate.Area.CURRENT})
@BlockFree
@SCJAllowed
public static Double  valueOf( double val )
```

Allocates a Double in caller’s scope.

B.2.15 CLASS Enum

DECLARATION

```
@SCJAllowed
public abstract class Enum
  implements java.lang.Comparable, java.io.Serializable
  extends java.lang.Object
```

CONSTRUCTORS

```
@BlockFree
@MemoryAreaEncloses(inner = {”this”}, outer = {”name”})
@SCJAllowed
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”.

METHODS

```
@Allocate({javax.safetycritical.annotate.Allocate.Area.CURRENT})
@BlockFree
@SCJAllowed
protected final Object  clone( )
```

Does not allow ”this” to escape local variables.

@BlockFree
@SCJAllowed
public final int **compareTo**(Enum o)

Allocates no memory. Does not allow ”this” or ”o” argument to escape local variables.

@BlockFree
@SCJAllowed
public final boolean **equals**(Object o)

Allocates no memory. Does not allow ”this” or ”o” argument to escape local variables.

@BlockFree
@SCJAllowed
public final Class **getDeclaringClass**( )

Allocates no memory. Returns a reference to a previously allocated Class, which resides in its ClassLoader scope.

@BlockFree
@SCJAllowed
public final int **hashCode**( )

Allocates no memory. Does not allow ”this” to escape local variables.

@BlockFree
@SCJAllowed
public final 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.

```
@BlockFree
@SCJAllowed
public final int ordinal( )
```

Allocates no memory. Does not allow "this" to escape local variables.

```
@Allocate({javax.safetycritical.annotate.Allocate.Area.CURRENT})
@BlockFree
@SCJAllowed
public String toString( )
```

Does not allow "this" to escape local variables. Allocates a String and associated internal "structure" (e.g. char[]) in caller's scope. (Note: this semantics is desired for consistency with overridden implementation of Object.toString()).

B.2.16 CLASS Error

DECLARATION

```
@SCJAllowed
public class Error
  implements java.io.Serializable
  extends java.lang.Throwable
```

CONSTRUCTORS

```
@Allocate({javax.safetycritical.annotate.Allocate.Area.CURRENT})
@BlockFree
@SCJAllowed
public Error( )
```

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).

```
@Allocate({javax.safetycritical.annotate.Allocate.Area.CURRENT})
@BlockFree
@MemoryAreaEncloses(inner = {”this”}, outer = {”msg”})
@SCJAllowed
public Error( 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).

```
@Allocate({javax.safetycritical.annotate.Allocate.Area.CURRENT})
@BlockFree
@MemoryAreaEncloses(inner = {”this”, ”this”}, outer = {”msg”, ”t”})
@SCJAllowed
public Error( 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).

```
@Allocate({javax.safetycritical.annotate.Allocate.Area.CURRENT})
@BlockFree
@MemoryAreaEncloses(inner = {”this”}, outer = {”t”})
@SCJAllowed
public Error( Throwable t )
```

Shall not copy ”this” to any instance or static field. The scope containing the msg argument must enclose the scope containing ”this”. Otherwise, an IllegalAssignmentError will be thrown.

Allocates an application- and implementation-dependent amount of memory in the current scope (to represent stack backtrace).

B.2.17 CLASS **Exception**

DECLARATION

```
@SCJAllowed
public class Exception
  implements java.io.Serializable
  extends java.lang.Throwable
```

CONSTRUCTORS

```
@Allocate({javax.safetycritical.annotate.Allocate.Area.CURRENT})
@BlockFree
@SCJAllowed
public Exception( )
```

Shall not copy ”this” to any instance or static field.

Invokes System.captureStackBacktrace(this) to save the back trace associated with the current thread.

```
@Allocate({javax.safetycritical.annotate.Allocate.Area.CURRENT})
@BlockFree
@MemoryAreaEncloses(inner = {”this”}, outer = {”msg”})
@SCJAllowed
public Exception( String msg )
```

Shall not copy ”this” to any instance or static field. The scope containing the msg argument must enclose the scope containing ”this”. Otherwise, an IllegalAssignmentError will be thrown.

Invokes System.captureStackBacktrace(this) to save the back trace associated with the current thread.

```
@BlockFree
@MemoryAreaEncloses(inner = {”this”}, outer = {”cause”})
@SCJAllowed
public Exception( Throwable cause )
```

Shall not copy ”this” to any instance or static field.

Does not invoke System.captureStackBacktrace(this) so as to not overwrite the backtrace associated with cause.

```
@BlockFree
@MemoryAreaEncloses(inner = {”this”, ”this”}, outer = {”cause”, ”msg”})
@SCJAllowed
public Exception( String msg , Throwable cause )
```

Shall not copy ”this” to any instance or static field.

Does not invoke System.captureStackBacktrace(this) so as to not overwrite the backtrace associated with cause.

B.2.18 CLASS ExceptionInInitializerError

DECLARATION

```
@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.2.19 CLASS Float

DECLARATION

```
@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 Class TYPE

CONSTRUCTORS

@BlockFree
@SCJAllowed
public **Float**(float val)

Allocates no memory. Does not allow ”this” to escape local variables.

@BlockFree
@SCJAllowed
public **Float**(double val)

Allocates no memory. Does not allow ”this” to escape local variables.

@BlockFree
@SCJAllowed
public **Float**(String str)

Allocates no memory. Does not allow ”this” or ”str” argument to escape local variables.

METHODS

@BlockFree
@SCJAllowed
public byte **byteValue**( )

Allocates no memory. Does not allow ”this” to escape local variables.

@BlockFree
@SCJAllowed
public static int **compare**(float value1 , float value2)

Allocates no memory.

@BlockFree
@SCJAllowed
public int **compareTo**(Float other)

Allocates no memory. Does not allow ”this” or ”other” argument to escape local variables.

@BlockFree
@SCJAllowed
public double **doubleValue**( )

Allocates no memory. Does not allow ”this” to escape local variables.

@BlockFree
@SCJAllowed
public boolean **equals**(Object obj)

Allocates no memory. Does not allow ”this” or ”obj” argument to escape local variables.

@BlockFree
@SCJAllowed
public static int **floatToIntBits**(float v)

Allocates no memory.

@BlockFree
@SCJAllowed
public static int **floatToRawIntBits**(float v)

Allocates no memory.

@BlockFree
@SCJAllowed
public float **floatValue**( )

Allocates no memory. Does not allow ”this” to escape local variables.

@BlockFree
@SCJAllowed
public int **hashCode**( )

Allocates no memory. Does not allow ”this” to escape local variables.

@BlockFree
@SCJAllowed
public static float **intBitsToFloat**(int v)

Allocates no memory.

@BlockFree
@SCJAllowed
public int **intValue**( )

Allocates no memory. Does not allow ”this” to escape local variables.

@BlockFree
@SCJAllowed
public static boolean **isInfinite**(float v)

Allocates no memory.

@BlockFree
@SCJAllowed
public boolean **isInfinite**( )

Allocates no memory. Does not allow ”this” to escape local variables.

@BlockFree
@SCJAllowed
public static boolean **isNaN**(float v)

Allocates no memory.

@BlockFree
@SCJAllowed
public boolean **isNaN**( )

Allocates no memory. Does not allow ”this” to escape local variables.

@BlockFree
@SCJAllowed
public long **longValue**( )

Allocates no memory. Does not allow ”this” to escape local variables.

```
@BlockFree
@SCJAllowed
public static float parseFloat( String s )
```

Allocates no memory. Does not allow ”s” argument to escape local variables.

```
@BlockFree
@SCJAllowed
public short shortValue( )
```

Allocates no memory. Does not allow ”this” to escape local variables.

```
@Allocate({javax.safetycritical.annotate.Allocate.Area.CURRENT})
@BlockFree
@SCJAllowed
public static String toHexString( float v )
```

Allocates a String and associated internal ”structure” (e.g. char[]) in caller’s scope.

```
@Allocate({javax.safetycritical.annotate.Allocate.Area.CURRENT})
@BlockFree
@SCJAllowed
public static String toString( float v )
```

Allocates a String and associated internal ”structure” (e.g. char[]) in caller’s scope.

```
@Allocate({javax.safetycritical.annotate.Allocate.Area.CURRENT})
@BlockFree
@SCJAllowed
public 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()).

```
@BlockFree
@SCJAllowed
public static Float  valueOf( String str )
```

Does not allow ”this” to escape local variables. Allocates a Float in caller’s scope.

```
@Allocate({javax.safetycritical.annotate.Allocate.Area.CURRENT})
@BlockFree
@SCJAllowed
public static Float  valueOf( float val )
```

Allocates a Float in caller’s scope.

B.2.20 CLASS IllegalArgumentException

DECLARATION

```
@SCJAllowed
public class IllegalArgumentException
  implements java.io.Serializable
  extends java.lang.RuntimeException
```

CONSTRUCTORS

```
@Allocate({javax.safetycritical.annotate.Allocate.Area.CURRENT})
@BlockFree
@SCJAllowed
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).

```
@Allocate({javax.safetycritical.annotate.Allocate.Area.CURRENT})
@BlockFree
@MemoryAreaEncloses(inner = {"this"}, outer = {"msg"})
@SCJAllowed
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).

```
@Allocate({javax.safetycritical.annotate.Allocate.Area.CURRENT})
@BlockFree
@MemoryAreaEncloses(inner = {"this", "this"}, outer = {"msg", "t"})
@SCJAllowed
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).

```
@Allocate({javax.safetycritical.annotate.Allocate.Area.CURRENT})
@BlockFree
@MemoryAreaEncloses(inner = {"this"}, outer = {"t"})
@SCJAllowed
public IllegalArgumentException( Throwable t )
```

Shall not copy "this" to any instance or static field. The scope containing the msg argument must enclose the scope containing "this". Otherwise, an IllegalAssignmentError will be thrown.

Allocates an application- and implementation-dependent amount of memory in the current scope (to represent stack backtrace).

B.2.21 CLASS **IllegalMonitorStateException**

DECLARATION

```
@SCJAllowed(javax.safetycritical.annotate.Level.LEVEL_2)
public class IllegalMonitorStateException
  implements java.io.Serializable
  extends java.lang.RuntimeException
```

CONSTRUCTORS

```
@Allocate({javax.safetycritical.annotate.Allocate.Area.CURRENT})
@BlockFree
@SCJAllowed
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).

```
@Allocate({javax.safetycritical.annotate.Allocate.Area.CURRENT})
@BlockFree
@MemoryAreaEncloses(inner = {”this”}, outer = {”msg”})
@SCJAllowed
public IllegalMonitorStateException( String msg )
```

Shall not copy ”this” to any instance or static field. The scope containing the msg argument must enclose the scope containing ”this”. Otherwise, an IllegalAssignmentError will be thrown.

Allocates an application- and implementation-dependent amount of memory in the current scope (to represent stack backtrace).

```
@Allocate({javax.safetycritical.annotate.Allocate.Area.CURRENT})
@BlockFree
@MemoryAreaEncloses(inner = {”this”, ”this”}, outer = {”msg”, ”t”})
@SCJAllowed
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).

```
@Allocate({javax.safetycritical.annotate.Allocate.Area.CURRENT})
@BlockFree
@MemoryAreaEncloses(inner = {”this”}, outer = {”t”})
@SCJAllowed
public IllegalMonitorStateException( Throwable t )
```

Shall not copy ”this” to any instance or static field. The scope containing the msg argument must enclose the scope containing ”this”. Otherwise, an IllegalAssignmentError will be thrown.

Allocates an application- and implementation-dependent amount of memory in the current scope (to represent stack backtrace).

B.2.22 CLASS IllegalStateException

DECLARATION

```
@SCJAllowed
public class IllegalStateException
  implements java.io.Serializable
  extends java.lang.RuntimeException
```

CONSTRUCTORS

```
@Allocate({javax.safetycritical.annotate.Allocate.Area.CURRENT})
@BlockFree
@SCJAllowed
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).

```
@Allocate({javax.safetycritical.annotate.Allocate.Area.CURRENT})
@BlockFree
@MemoryAreaEncloses(inner = {"this"}, outer = {"msg"})
@SCJAllowed
public IllegalStateException( String msg )
```

Shall not copy "this" to any instance or static field. The scope containing the msg argument must enclose the scope containing "this". Otherwise, an IllegalAssignmentError will be thrown.

Allocates an application- and implementation-dependent amount of memory in the current scope (to represent stack backtrace).

B.2.23 CLASS IllegalThreadStateException

DECLARATION

```
@SCJAllowed
public class IllegalThreadStateException
  implements java.io.Serializable
  extends java.lang.RuntimeException
```

CONSTRUCTORS

```
@Allocate({javax.safetycritical.annotate.Allocate.Area.CURRENT})
@BlockFree
@SCJAllowed
public IllegalThreadStateException( )
```

```
@Allocate({javax.safetycritical.annotate.Allocate.Area.CURRENT})
@BlockFree
@MemoryAreaEncloses(inner = {"this"}, outer = {"msg"})
@SCJAllowed
public IllegalThreadStateException( String description )
```

B.2.24 CLASS **IncompatibleClassChangeError**

DECLARATION

```
@SCJAllowed(javax.safetycritical.annotate.Level.LEVEL_2)
public class IncompatibleClassChangeError
  implements java.io.Serializable
  extends java.lang.RuntimeException
```

CONSTRUCTORS

```
@Allocate({javax.safetycritical.annotate.Allocate.Area.CURRENT})
@BlockFree
@SCJAllowed
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).

```
@Allocate({javax.safetycritical.annotate.Allocate.Area.CURRENT})
@BlockFree
@MemoryAreaEncloses(inner = {”this”}, outer = {”msg”})
@SCJAllowed
public IncompatibleClassChangeError( String msg )
```

Shall not copy ”this” to any instance or static field. The scope containing the msg argument must enclose the scope containing ”this”. Otherwise, an IllegalAssignmentError will be thrown.

Allocates an application- and implementation-dependent amount of memory in the current scope (to represent stack backtrace).

B.2.25 CLASS **IndexOutOfBoundsException**

DECLARATION

```
@SCJAllowed
public class IndexOutOfBoundsException
  implements java.io.Serializable
  extends java.lang.RuntimeException
```

CONSTRUCTORS

```
@Allocate({javax.safetycritical.annotate.Allocate.Area.CURRENT})
@BlockFree
@SCJAllowed
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).

```
@Allocate({javax.safetycritical.annotate.Allocate.Area.CURRENT})
@BlockFree
@MemoryAreaEncloses(inner = {”this”}, outer = {”msg”})
@SCJAllowed
public IndexOutOfBoundsException( String msg )
```

Shall not copy ”this” to any instance or static field. The scope containing the msg argument must enclose the scope containing ”this”. Otherwise, an IllegalAssignmentError will be thrown.

Allocates an application- and implementation-dependent amount of memory in the current scope (to represent stack backtrace).

B.2.26 CLASS InstantiationException

DECLARATION

```
@SCJAllowed
public class InstantiationException
  implements java.io.Serializable
  extends java.lang.Exception
```

CONSTRUCTORS

```
@Allocate({javax.safetycritical.annotate.Allocate.Area.CURRENT})
@BlockFree
@SCJAllowed
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).

```
@Allocate({javax.safetycritical.annotate.Allocate.Area.CURRENT})
@BlockFree
@MemoryAreaEncloses(inner = {”this”}, outer = {”msg”})
@SCJAllowed
public InstantiationException( String msg )
```

Shall not copy ”this” to any instance or static field. The scope containing the msg argument must enclose the scope containing ”this”. Otherwise, an IllegalAssignmentError will be thrown.

Allocates an application- and implementation-dependent amount of memory in the current scope (to represent stack backtrace).

B.2.27 CLASS **Integer**

DECLARATION

```
@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 Class TYPE
```

CONSTRUCTORS

```
@BlockFree
@SCJAllowed
public Integer( int val )
```

Allocates no memory. Does not allow ”this” to escape local variables.

```
@BlockFree
@SCJAllowed
public Integer( String str )
```

Allocates no memory. Does not allow ”this” or ”str” argument to escape local variables.

METHODS

@BlockFree
@SCJAllowed
public static int **bitCount**(int i)

Allocates no memory. Does not allow ”this” to escape local variables.

@BlockFree
@SCJAllowed
public byte **byteValue**( )

Allocates no memory. Does not allow ”this” to escape local variables.

@BlockFree
@SCJAllowed
public int **compareTo**(Integer other)

Allocates no memory. Does not allow ”this” or ”other” argument to escape local variables.

@BlockFree
@SCJAllowed
public static Integer **decode**(String str)

Does not allow ”str” argument to escape local variables. Allocates an Integer in caller’s scope.

@BlockFree
@SCJAllowed
public double **doubleValue**( )

Allocates no memory.

```
@BlockFree
@SCJAllowed
public boolean  equals( Object obj )
```

Allocates no memory. Does not allow ”this” or ”obj” argument to escape local variables.

```
@BlockFree
@SCJAllowed
public float  floatValue( )
```

Allocates no memory. Does not allow ”this” to escape local variables.

```
@Allocate({javax.safetycritical.annotate.Allocate.Area.CURRENT})
@BlockFree
@SCJAllowed
public static Integer  getInteger( String str , Integer v )
```

Does not allow ”str” or ”v” arguments to escape local variables. Allocates Integer in caller’s scope.

```
@Allocate({javax.safetycritical.annotate.Allocate.Area.CURRENT})
@BlockFree
@SCJAllowed
public static Integer  getInteger( String str , int v )
```

Does not allow ”str” argument to escape local variables. Allocates Integer in caller’s scope.

```
@Allocate({javax.safetycritical.annotate.Allocate.Area.CURRENT})
@BlockFree
@SCJAllowed
public static Integer  getInteger( String str )
```

Does not allow ”str” argument to escape local variables. Allocates Integer in caller’s scope.

@BlockFree
@SCJAllowed
public int **hashCode**( )

Allocates no memory. Does not allow ”this” to escape local variables.

@BlockFree
@SCJAllowed
public static int **highestOneBit**(int i)

Allocates no memory.

@BlockFree
@SCJAllowed
public int **intValue**( )

Allocates no memory. Does not allow ”this” to escape local variables.

@BlockFree
@SCJAllowed
public long **longValue**( )

Allocates no memory. Does not allow ”this” to escape local variables.

@BlockFree
@SCJAllowed
public static int **lowestOneBit**(int i)

Allocates no memory.

@BlockFree
@SCJAllowed
public static int **numberOfLeadingZeros**(int i)

Allocates no memory.

@BlockFree
@SCJAllowed
public static int **parseInt**(String str , int radix)

Allocates no memory. Does not allow ”str” argument to escape local variables.

@BlockFree
@SCJAllowed
public static int **parseInt**(String str)

Allocates no memory. Does not allow ”str” argument to escape local variables.

@BlockFree
@SCJAllowed
public static int **reverse**(int i)

Allocates no memory.

@BlockFree
@SCJAllowed
public static int **reverseBytes**(int i)

Allocates no memory.

@BlockFree
@SCJAllowed
public static int **rotateLeft**(int i , int distance)

Allocates no memory.

@BlockFree
@SCJAllowed
public static int **rotateRight**(int i , int distance)

Allocates no memory.

@BlockFree
@SCJAllowed
public short **shortValue**( )

Allocates no memory. Does not allow ”this” to escape local variables.

@BlockFree
@SCJAllowed
public static int **sigNum**(int i)

Allocates no memory.

@Allocate({javax.safetycritical.annotate.Allocate.Area.CURRENT})
@BlockFree
@SCJAllowed
public static 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.

@Allocate({javax.safetycritical.annotate.Allocate.Area.CURRENT})
@BlockFree
@SCJAllowed
public static 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.

```
@Allocate({javax.safetycritical.annotate.Allocate.Area.CURRENT})
@BlockFree
@SCJAllowed
public static 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.

```
@Allocate({javax.safetycritical.annotate.Allocate.Area.CURRENT})
@BlockFree
@SCJAllowed
public static String toString( int v )
```

Allocates a String and associated internal ”structure” (e.g. char[]) in caller’s scope.

```
@Allocate({javax.safetycritical.annotate.Allocate.Area.CURRENT})
@BlockFree
@SCJAllowed
public static 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.

```
@Allocate({javax.safetycritical.annotate.Allocate.Area.CURRENT})
@BlockFree
@SCJAllowed
public 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()).

@Allocate({javax.safetycritical.annotate.Allocate.Area.CURRENT})
@BlockFree
@SCJAllowed
public static Integer **valueOf**(String str , int base)

Does not allow ”str” argument to escape local variables. Allocates an Integer in caller’s scope.

@Allocate({javax.safetycritical.annotate.Allocate.Area.CURRENT})
@BlockFree
@SCJAllowed
public static Integer **valueOf**(String str)

Does not allow ”str” argument to escape local variables. Allocates an Integer in caller’s scope.

@Allocate({javax.safetycritical.annotate.Allocate.Area.CURRENT})
@BlockFree
@SCJAllowed
public static Integer **valueOf**(int val)

Allocates an Integer in caller’s scope.

B.2.28 CLASS **InternalError**

DECLARATION

```
@SCJAllowed
public class InternalError
  implements java.io.Serializable
  extends java.lang.VirtualMachineError
```

CONSTRUCTORS

@Allocate({javax.safetycritical.annotate.Allocate.Area.CURRENT})
@BlockFree

```
@SCJAllowed
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).

```
@Allocate({javax.safetycritical.annotate.Allocate.Area.CURRENT})
@BlockFree
@MemoryAreaEncloses(inner = {”this”}, outer = {”msg”})
@SCJAllowed
public InternalError( String msg )
```

Shall not copy ”this” to any instance or static field. The scope containing the msg argument must enclose the scope containing ”this”. Otherwise, an IllegalAssignmentError will be thrown.

Allocates an application- and implementation-dependent amount of memory in the current scope (to represent stack backtrace).

B.2.29 CLASS InterruptedException

DECLARATION

```
@SCJAllowed
public class InterruptedException
  implements java.io.Serializable
  extends java.lang.Exception
```

CONSTRUCTORS

```
@Allocate({javax.safetycritical.annotate.Allocate.Area.CURRENT})
@BlockFree
@SCJAllowed
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).

```
@Allocate({javax.safetycritical.annotate.Allocate.Area.CURRENT})
@BlockFree
@MemoryAreaEncloses(inner = {”this”}, outer = {”msg”})
@SCJAllowed
public InterruptedException( String msg )
```

Shall not copy ”this” to any instance or static field. The scope containing the msg argument must enclose the scope containing ”this”. Otherwise, an IllegalAssignmentError will be thrown.

Allocates an application- and implementation-dependent amount of memory in the current scope (to represent stack backtrace).

B.2.30 CLASS InvocationTargetException

DECLARATION

```
@SCJAllowed
public class InvocationTargetException
  extends java.lang.Exception
```

CONSTRUCTORS

```
@SCJAllowed
public InvocationTargetException( )
```

```
@SCJAllowed
public InvocationTargetException( String msg )
```

```
@SCJAllowed
public InvocationTargetException( Throwable cause )
```

@SCJAllowed
public **InvocationTargetException**(String msg , Throwable cause)

B.2.31 CLASS **Long**

DECLARATION

```
@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 Class TYPE

CONSTRUCTORS

@BlockFree
@SCJAllowed
public **Long**(long val)

Allocates no memory. Does not allow ”this” to escape local variables.

@BlockFree
@SCJAllowed
public **Long**(String str)

Allocates no memory. Does not allow ”this” or ”str” argument to escape local variables.

METHODS

@BlockFree
@SCJAllowed
public static int **bitCount**(long i)

Allocates no memory.

@BlockFree
@SCJAllowed
public byte **byteValue**( )

Allocates no memory. Does not allow ”this” to escape local variables.

@BlockFree
@SCJAllowed
public int **compareTo**(Long other)

Allocates no memory. Does not allow ”this” or ”other” argument to escape local variables.

@BlockFree
@SCJAllowed
public static Long **decode**(String str)

Does not allow ”str” argument to escape local variables. Allocates a Long result object in the caller’s scope.

@BlockFree
@SCJAllowed
public double **doubleValue**( )

Allocates no memory. Does not allow ”this” to escape local variables.

@BlockFree
@SCJAllowed
public boolean **equals**(Object obj)

Allocates no memory. Does not allow ”this” or ”obj” argument to escape local variables.

@BlockFree
@SCJAllowed
public float **floatValue**( )

Allocates no memory. Does not allow ”this” to escape local variables.

@BlockFree
@SCJAllowed
public static 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.

@BlockFree
@SCJAllowed
public static 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.

@BlockFree
@SCJAllowed
public static Long **getLong**(String str)

Does not allow ”str” argument to escape local variables. Allocates a Long result object in the caller’s scope.

@BlockFree
@SCJAllowed
public int **hashCode**( )

Allocates no memory. Does not allow ”this” to escape local variables.

@BlockFree
@SCJAllowed
public static long **highestOneBit**(long i)

Allocates no memory.

@BlockFree
@SCJAllowed
public int **intValue**( )

Allocates no memory. Does not allow ”this” to escape local variables.

@BlockFree
@SCJAllowed
public long **longValue**( )

Allocates no memory. Does not allow ”this” to escape local variables.

@BlockFree
@SCJAllowed
public static long **lowestOneBit**(long i)

Allocates no memory.

@BlockFree
@SCJAllowed
public static int **numberOfLeadingZeros**(long i)

Allocates no memory.

@BlockFree
@SCJAllowed
public static int **numberOfTrailingZeros**(long i)

Allocates no memory.

@BlockFree
@SCJAllowed
public static long **parseLong**(String str , int base)

Allocates no memory. Does not allow ”this” or ”other” argument to escape local variables.

@BlockFree
@SCJAllowed
public static long **parseLong**(String str)

Allocates no memory. Does not allow ”this” or ”other” argument to escape local variables.

@BlockFree
@SCJAllowed
public static long **reverse**(long i)

Allocates no memory.

@BlockFree
@SCJAllowed
public static long **reverseBytes**(long i)

Allocates no memory.

@BlockFree
@SCJAllowed
public static long **rotateLeft**(long i , int distance)

Allocates no memory.

@BlockFree
@SCJAllowed
public static long **rotateRight**(long i , int distance)

Allocates no memory.

@BlockFree
@SCJAllowed
public short **shortValue**( )

Allocates no memory. Does not allow ”this” to escape local variables.

@BlockFree
@SCJAllowed
public static int **signum**(long i)

Allocates no memory.

@Allocate({javax.safetycritical.annotate.Allocate.Area.CURRENT})
@BlockFree
@SCJAllowed
public static 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.

```
@Allocate({javax.safetycritical.annotate.Allocate.Area.CURRENT})
@BlockFree
@SCJAllowed
public static 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.

```
@Allocate({javax.safetycritical.annotate.Allocate.Area.CURRENT})
@BlockFree
@SCJAllowed
public static 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.

```
@Allocate({javax.safetycritical.annotate.Allocate.Area.CURRENT})
@BlockFree
@SCJAllowed
public static String toString( long v )
```

Allocates a String and associated internal ”structure” (e.g. char[]) in caller’s scope.

```
@Allocate({javax.safetycritical.annotate.Allocate.Area.CURRENT})
@BlockFree
@SCJAllowed
public static String toString( long v , int base )
```

Allocates a String and associated internal ”structure” (e.g. char[]) in caller’s scope.

@Allocate({javax.safetycritical.annotate.Allocate.Area.CURRENT})
@BlockFree
@SCJAllowed
public 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()).

@Allocate({javax.safetycritical.annotate.Allocate.Area.CURRENT})
@BlockFree
@SCJAllowed
public static Long **valueOf**(String str , int base)

Does not allow ”str” argument to escape local variables. Allocates a Long in caller’s scope.

@Allocate({javax.safetycritical.annotate.Allocate.Area.CURRENT})
@BlockFree
@SCJAllowed
public static Long **valueOf**(String str)

Does not allow ”str” argument to escape local variables. Allocates a Long in caller’s scope.

@Allocate({javax.safetycritical.annotate.Allocate.Area.CURRENT})
@BlockFree
@SCJAllowed
public static Long **valueOf**(long val)

Allocates a Long in caller’s scope.

B.2.32 CLASS **Math**

DECLARATION

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

FIELDS

```
 @SCJAllowed
public static final double E
```

```
 @SCJAllowed
public static final double PI
```

METHODS

```
@BlockFree
@SCJAllowed
public static double  IEEEremainder( double f1 , double f2 )
```

Allocates no memory.

```
@BlockFree
@SCJAllowed
public static long  abs( long a )
```

Allocates no memory.

```
@BlockFree
@SCJAllowed
public static double  abs( double a )
```

Allocates no memory.

@BlockFree
@SCJAllowed
public static float **abs**(float a)

Allocates no memory.

@BlockFree
@SCJAllowed
public static int **abs**(int a)

Allocates no memory.

@BlockFree
@SCJAllowed
public static double **acos**(double a)

Allocates no memory.

@BlockFree
@SCJAllowed
public static double **asin**(double a)

Allocates no memory.

@BlockFree
@SCJAllowed
public static double **atan**(double a)

Allocates no memory.

@BlockFree
@SCJAllowed
public static double **atan2**(double a , double b)

Allocates no memory.

@BlockFree
@SCJAllowed
public static double **cbrt**(double a)

Allocates no memory.

@BlockFree
@SCJAllowed
public static double **ceil**(double a)

Allocates no memory.

@BlockFree
@SCJAllowed
public static double **copySign**(float magnitude , float sign)

Allocates no memory.

@BlockFree
@SCJAllowed
public static double **copySign**(double magnitude , double sign)

Allocates no memory.

@BlockFree
@SCJAllowed
public static double **cos**(double a)

Allocates no memory.

@BlockFree
@SCJAllowed
public static double **cosh**(double a)

Allocates no memory.

@BlockFree
@SCJAllowed
public static double **exp**(double a)

Allocates no memory.

@BlockFree
@SCJAllowed
public static double **expm1**(double a)

Allocates no memory.

@BlockFree
@SCJAllowed
public static double **floor**(double a)

Allocates no memory.

@BlockFree
@SCJAllowed
public static int **getExponent**(float a)

Allocates no memory.

@BlockFree
@SCJAllowed
public static int **getExponent**(double a)

Allocates no memory.

@BlockFree
@SCJAllowed
public static double **hypot**(double x , double y)

Allocates no memory.

@BlockFree
@SCJAllowed
public static double **log**(double a)

Allocates no memory.

@BlockFree
@SCJAllowed
public static double **log10**(double a)

Allocates no memory.

@BlockFree
@SCJAllowed
public static double **log1p**(double a)

Allocates no memory.

@BlockFree
@SCJAllowed
public static double **max**(double a , double b)

Allocates no memory.

@BlockFree
@SCJAllowed
public static long **max**(long a , long b)

Allocates no memory.

@BlockFree
@SCJAllowed
public static float **max**(float a , float b)

Allocates no memory.

@BlockFree
@SCJAllowed
public static int **max**(int a , int b)

Allocates no memory.

@BlockFree
@SCJAllowed
public static long **min**(long a , long b)

Allocates no memory.

@BlockFree
@SCJAllowed
public static double **min**(double a , double b)

Allocates no memory.

@BlockFree
@SCJAllowed
public static float **min**(float a , float b)

Allocates no memory.

@BlockFree
@SCJAllowed
public static int **min**(int a , int b)

Allocates no memory.

@BlockFree
@SCJAllowed
public static float **nextAfter**(float start , float direction)

Allocates no memory.

@BlockFree
@SCJAllowed
public static double **nextAfter**(double start , double direction)

Allocates no memory.

@BlockFree
@SCJAllowed
public static float **nextUp**(float d)

Allocates no memory.

@BlockFree
@SCJAllowed
public static double **nextUp**(double d)

Allocates no memory.

@BlockFree
@SCJAllowed
public static double **pow**(double a , double b)

Allocates no memory.

@BlockFree
@SCJAllowed
public static double **random**( )

Allocates no memory.

@BlockFree
@SCJAllowed
public static double **rint**(double a)

Allocates no memory.

@BlockFree
@SCJAllowed
public static long **round**(double a)

Allocates no memory.

@BlockFree
@SCJAllowed
public static int **round**(float a)

Allocates no memory.

@BlockFree
@SCJAllowed
public static float **scalb**(float f , int scaleFactor)

Allocates no memory.

@BlockFree
@SCJAllowed
public static double **scalb**(double d , int scaleFactor)

Allocates no memory.

@BlockFree
@SCJAllowed
public static float **signum**(float f)

Allocates no memory.

@BlockFree
@SCJAllowed
public static double **signum**(double d)

Allocates no memory.

@BlockFree
@SCJAllowed
public static double **sin**(double a)

Allocates no memory.

@BlockFree
@SCJAllowed
public static double **sinh**(double a)

Allocates no memory.

@BlockFree
@SCJAllowed
public static double **sqrt**(double a)

Allocates no memory.

@BlockFree
@SCJAllowed
public static double **tan**(double a)

Allocates no memory.

@BlockFree
@SCJAllowed
public static double **tanh**(double a)

Allocates no memory.

@BlockFree
@SCJAllowed
public static double **toDegrees**(double val)

Allocates no memory.

@BlockFree
@SCJAllowed
public static double **toRadians**(double val)

Allocates no memory.

```
@BlockFree
@SCJAllowed
public static float  ulp( float d )
```

Allocates no memory.

```
@BlockFree
@SCJAllowed
public static double  ulp( double d )
```

Allocates no memory.

B.2.33 CLASS NegativeArraySizeException

DECLARATION

```
@SCJAllowed
public class NegativeArraySizeException
  implements java.io.Serializable
  extends java.lang.RuntimeException
```

CONSTRUCTORS

```
@Allocate({javax.safetycritical.annotate.Allocate.Area.CURRENT})
@BlockFree
@SCJAllowed
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).

```
@Allocate({javax.safetycritical.annotate.Allocate.Area.CURRENT})
@BlockFree
@MemoryAreaEncloses(inner = {”this”}, outer = {”msg”})
```

```
@SCJAllowed
public NegativeArraySizeException( String msg )
```

Shall not copy ”this” to any instance or static field. The scope containing the msg argument must enclose the scope containing ”this”. Otherwise, an IllegalAssignmentError will be thrown.

Allocates an application- and implementation-dependent amount of memory in the current scope (to represent stack backtrace).

B.2.34 CLASS NullPointerException

DECLARATION

```
@SCJAllowed
public class NullPointerException
  implements java.io.Serializable
  extends java.lang.RuntimeException
```

CONSTRUCTORS

```
@Allocate({javax.safetycritical.annotate.Allocate.Area.CURRENT})
@BlockFree
@SCJAllowed
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).

```
@Allocate({javax.safetycritical.annotate.Allocate.Area.CURRENT})
@BlockFree
@MemoryAreaEncloses(inner = {”this”}, outer = {”msg”})
@SCJAllowed
public NullPointerException( String msg )
```

Shall not copy ”this” to any instance or static field. The scope containing the msg argument must enclose the scope containing ”this”. Otherwise, an IllegalAssignmentError will be thrown.

Allocates an application- and implementation-dependent amount of memory in the current scope (to represent stack backtrace).

B.2.35 CLASS **Number**

DECLARATION

```
@SCJAllowed
public abstract class Number
  implements java.io.Serializable
  extends java.lang.Object
```

CONSTRUCTORS

```
@BlockFree
@SCJAllowed
public Number( )
```

The implementation of this method shall not allow ”this” to escape the method’s local variables.

METHODS

```
@BlockFree
@SCJAllowed
public byte  byteValue( )
```

The implementation of this method shall not allow ”this” to escape the method’s local variables.

```
@BlockFree
@SCJAllowed
public abstract double  doubleValue( )
```

The implementation of this method shall not allow ”this” to escape the method’s local variables.

```
@BlockFree
@SCJAllowed
public abstract float floatValue( )
```

The implementation of this method shall not allow ”this” to escape the method’s local variables.

```
@BlockFree
@SCJAllowed
public abstract int intValue( )
```

The implementation of this method shall not allow ”this” to escape the method’s local variables.

```
@BlockFree
@SCJAllowed
public abstract long longValue( )
```

The implementation of this method shall not allow ”this” to escape the method’s local variables.

```
@BlockFree
@SCJAllowed
public abstract short shortValue( )
```

The implementation of this method shall not allow ”this” to escape the method’s local variables.

B.2.36 CLASS **NumberFormatException**

DECLARATION

```
@SCJAllowed
public class NumberFormatException
  implements java.io.Serializable
  extends java.lang.IllegalArgumentException
```

CONSTRUCTORS

```
@Allocate({javax.safetycritical.annotate.Allocate.Area.CURRENT})
@BlockFree
@SCJAllowed
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).

```
@Allocate({javax.safetycritical.annotate.Allocate.Area.CURRENT})
@BlockFree
@MemoryAreaEncloses(inner = {”this”}, outer = {”msg”})
@SCJAllowed
public NumberFormatException( String msg )
```

Shall not copy ”this” to any instance or static field. The scope containing the msg argument must enclose the scope containing ”this”. Otherwise, an IllegalAssignmentError will be thrown.

Allocates an application- and implementation-dependent amount of memory in the current scope (to represent stack backtrace).

B.2.37 CLASS **Object**

DECLARATION

```
@SCJAllowed
public class Object
```

CONSTRUCTORS

@BlockFree
@SCJAllowed
public **Object**( )

Allocates no memory. Does not allow ”this” to escape local variables.

METHODS

@BlockFree
@SCJAllowed
public boolean **equals**(Object obj)

Allocates no memory. Does not allow ”this” or ”obj” argument to escape local variables.

@BlockFree
@SCJAllowed
public final Class **getClass**( )

Allocates no memory. Does not allow ”this” or ”obj” argument to escape local variables.

@BlockFree
@SCJAllowed
public int **hashCode**( )

Allocates no memory. Does not allow ”this” to escape local variables.

@BlockFree
@SCJAllowed(javax.safetycritical.annotate.Level.LEVEL_2)
public final void **notify**( )

Allocates no memory. Does not allow ”this” to escape local variables.

```
@BlockFree
@SCJAllowed(javax.safetycritical.annotate.Level.LEVEL_2)
public final void notifyAll( )
```

Allocates no memory. Does not allow ”this” to escape local variables.

```
@Allocate({javax.safetycritical.annotate.Allocate.Area.CURRENT})
@BlockFree
@SCJAllowed
public String toString( )
```

Does not allow ”this” to escape local variables. Allocates a String and associated internal ”structure” (e.g. char[]) in caller’s scope.

```
@SCJAllowed(javax.safetycritical.annotate.Level.LEVEL_2)
public final void wait( long timeout , int nanos )
```

Allocates no memory. Does not allow ”this” to escape local variables.

```
@SCJAllowed(javax.safetycritical.annotate.Level.LEVEL_2)
public final void wait( long timeout )
```

Allocates no memory. Does not allow ”this” to escape local variables.

```
@SCJAllowed(javax.safetycritical.annotate.Level.LEVEL_2)
public final void wait( )
```

Allocates no memory. Does not allow ”this” to escape local variables.

B.2.38 CLASS **OutOfMemoryError**

DECLARATION

```
@SCJAllowed
public class OutOfMemoryError
  implements java.io.Serializable
  extends java.lang.VirtualMachineError
```

CONSTRUCTORS

```
@Allocate({javax.safetycritical.annotate.Allocate.Area.CURRENT})
@BlockFree
@SCJAllowed
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).

```
@Allocate({javax.safetycritical.annotate.Allocate.Area.CURRENT})
@BlockFree
@MemoryAreaEncloses(inner = {”this”}, outer = {”msg”})
@SCJAllowed
public OutOfMemoryError( String msg )
```

Shall not copy ”this” to any instance or static field. The scope containing the msg argument must enclose the scope containing ”this”. Otherwise, an IllegalAssignmentError will be thrown.

Allocates an application- and implementation-dependent amount of memory in the current scope (to represent stack backtrace).

B.2.39 CLASS **RuntimeException**

DECLARATION

```
@SCJAllowed
public class RuntimeException
  implements java.io.Serializable
  extends java.lang.Exception
```

CONSTRUCTORS

```
@Allocate({javax.safetycritical.annotate.Allocate.Area.CURRENT})
@BlockFree
@SCJAllowed
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).

```
@Allocate({javax.safetycritical.annotate.Allocate.Area.CURRENT})
@BlockFree
@MemoryAreaEncloses(inner = {”this”}, outer = {”msg”})
@SCJAllowed
public RuntimeException( String msg )
```

Shall not copy ”this” to any instance or static field. The scope containing the msg argument must enclose the scope containing ”this”. Otherwise, an IllegalAssignmentError will be thrown.

Allocates an application- and implementation-dependent amount of memory in the current scope (to represent stack backtrace).

```
@Allocate({javax.safetycritical.annotate.Allocate.Area.CURRENT})
@BlockFree
@MemoryAreaEncloses(inner = {”this”, ”this”}, outer = {”msg”, ”t”})
@SCJAllowed
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).

```
@Allocate({javax.safetycritical.annotate.Allocate.Area.CURRENT})
@BlockFree
@MemoryAreaEncloses(inner = {”this”}, outer = {”t”})
@SCJAllowed
public RuntimeException( Throwable t )
```

Shall not copy ”this” to any instance or static field. The scope containing the msg argument must enclose the scope containing ”this”. Otherwise, an IllegalAssignmentError will be thrown.

Allocates an application- and implementation-dependent amount of memory in the current scope (to represent stack backtrace).

B.2.40 CLASS Short

DECLARATION

```
@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 Class TYPE

CONSTRUCTORS

@BlockFree
@SCJAllowed
public **Short**(short val)

Allocates no memory. Does not allow ”this” to escape local variables.

@BlockFree
@SCJAllowed
public **Short**(String str)

Allocates no memory. Does not allow ”this” or ”str” argument to escape local variables.

METHODS

@BlockFree
@SCJAllowed
public byte **byteValue**( )

Allocates no memory. Does not allow ”this” to escape local variables.

@BlockFree
@SCJAllowed
public int **compareTo**(Short other)

Allocates no memory. Does not allow ”this” or ”other” argument to escape local variables.

```
@BlockFree
@SCJAllowed
public static Short decode( String str )
```

Does not allow ”str” argument to escape local variables. Allocates a Short in caller’s scope.

```
@BlockFree
@SCJAllowed
public double doubleValue( )
```

Allocates no memory.

```
@BlockFree
@SCJAllowed
public boolean equals( Object obj )
```

Allocates no memory. Does not allow ”this” or ”obj” argument to escape local variables.

```
@BlockFree
@SCJAllowed
public float floatValue( )
```

Allocates no memory. Does not allow ”this” to escape local variables.

```
@BlockFree
@SCJAllowed
public int hashCode( )
```

Allocates no memory. Does not allow ”this” to escape local variables.

@BlockFree
@SCJAllowed
public int **intValue**( )

Allocates no memory. Does not allow ”this” to escape local variables.

@BlockFree
@SCJAllowed
public long **longValue**( )

Allocates no memory. Does not allow ”this” to escape local variables.

@BlockFree
@SCJAllowed
public static short **parseShort**(String str , int base)

Allocates no memory. Does not allow ”str” argument to escape local variables.

@BlockFree
@SCJAllowed
public static short **parseShort**(String str)

Allocates no memory. Does not allow ”str” argument to escape local variables.

@BlockFree
@SCJAllowed
public static short **reverseBytes**(short i)

Allocates no memory.

@BlockFree
@SCJAllowed
public short **shortValue**( )

Allocates no memory. Does not allow ”this” to escape local variables.

@Allocate({javax.safetycritical.annotate.Allocate.Area.CURRENT})
@BlockFree
@SCJAllowed
public static String **toString**(short v)

Allocates a String and associated internal ”structure” (e.g. char[]) in caller’s scope.

@Allocate({javax.safetycritical.annotate.Allocate.Area.CURRENT})
@BlockFree
@SCJAllowed
public 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()).

@Allocate({javax.safetycritical.annotate.Allocate.Area.CURRENT})
@BlockFree
@SCJAllowed
public static Short **valueOf**(String str , int base)

Does not allow ”str” argument to escape local variables. Allocates an Integer in caller’s scope.

@Allocate({javax.safetycritical.annotate.Allocate.Area.CURRENT})
@BlockFree
@SCJAllowed
public static Short **valueOf**(String str)

Does not allow ”str” argument to escape local variables. Allocates an Integer in caller’s scope.

```
@Allocate({javax.safetycritical.annotate.Allocate.Area.CURRENT})
@BlockFree
@SCJAllowed
public static Short  valueOf( short val )
```

Allocates a Short in caller’s scope.

B.2.41 CLASS StackOverflowError

DECLARATION

```
@SCJAllowed
public class StackOverflowError
  implements java.io.Serializable
  extends java.lang.VirtualMachineError
```

CONSTRUCTORS

```
@Allocate({javax.safetycritical.annotate.Allocate.Area.CURRENT})
@BlockFree
@SCJAllowed
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).

```
@Allocate({javax.safetycritical.annotate.Allocate.Area.CURRENT})
@BlockFree
@MemoryAreaEncloses(inner = {”this”}, outer = {”msg”})
@SCJAllowed
public StackOverflowError( String msg )
```

Shall not copy ”this” to any instance or static field. The scope containing the msg argument must enclose the scope containing ”this”. Otherwise, an IllegalAssignmentError will be thrown.

Allocates an application- and implementation-dependent amount of memory in the current scope (to represent stack backtrace).

B.2.42 CLASS **StackTraceElement**

DECLARATION

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

CONSTRUCTORS

```
@SCJAllowed
@BlockFree
@MemoryAreaEncloses(inner = {”this”, ”this”, ”this”}, outer = {”declaringClass”, ”methodName”, ”fileName”})
public StackTraceElement( String declaringClass , String methodName , String fileName , int lineNumber )
```

Shall not copy ”this” to any instance or static field.

METHODS

```
@BlockFree
@SCJAllowed
public boolean  equals( Object obj )
```

Allocates no memory. Does not allow ”this” or ”obj” argument to escape local variables.

```
@BlockFree
@SCJAllowed
public 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.

@BlockFree
@SCJAllowed
public 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.

@BlockFree
@SCJAllowed
public int **getLineNumber**( )

Performs no memory allocation.

@BlockFree
@SCJAllowed
public 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.

@BlockFree
@SCJAllowed
public int **hashCode**( )

Allocates no memory. Does not allow ”this” to escape local variables.

@BlockFree
@SCJAllowed
public boolean **isNativeMethod**( )

Allocates no memory. Does not allow ”this” to escape local variables.

```
@Allocate({javax.safetycritical.annotate.Allocate.Area.CURRENT})
@BlockFree
@SCJAllowed
public String  toString( )
```

Does not allow ”this” to escape local variables. Allocates a String and associated internal ”structure” (e.g. char[]) in caller’s scope. (Note: this semantics is desired for consistency with overridden implementation of Object.toString()).

B.2.43 CLASS **StrictMath**

DECLARATION

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

FIELDS

```
 @SCJAllowed
public static final double E
```

```
 @SCJAllowed
public static final double PI
```

METHODS

```
@BlockFree
@SCJAllowed
public static double  IEEEremainder( double f1 , double f2 )
```

Allocates no memory.

@BlockFree
@SCJAllowed
public static long **abs**(long a)

Allocates no memory.

@BlockFree
@SCJAllowed
public static double **abs**(double a)

Allocates no memory.

@BlockFree
@SCJAllowed
public static float **abs**(float a)

Allocates no memory.

@BlockFree
@SCJAllowed
public static int **abs**(int a)

Allocates no memory.

@BlockFree
@SCJAllowed
public static double **acos**(double a)

Allocates no memory.

@BlockFree
@SCJAllowed
public static double **asin**(double a)

Allocates no memory.

@BlockFree
@SCJAllowed
public static double **atan**(double a)

Allocates no memory.

@BlockFree
@SCJAllowed
public static double **atan2**(double a , double b)

Allocates no memory.

@BlockFree
@SCJAllowed
public static double **cbrt**(double a)

Allocates no memory.

@BlockFree
@SCJAllowed
public static double **ceil**(double a)

Allocates no memory.

@BlockFree
@SCJAllowed
public static double **copySign**(float magnitude , float sign)

Allocates no memory.

@BlockFree
@SCJAllowed
public static double **copySign**(double magnitude , double sign)

Allocates no memory.

@BlockFree
@SCJAllowed
public static double **cos**(double a)

Allocates no memory.

@BlockFree
@SCJAllowed
public static double **cosh**(double a)

Allocates no memory.

@BlockFree
@SCJAllowed
public static double **exp**(double a)

Allocates no memory.

@BlockFree
@SCJAllowed
public static double **expm1**(double a)

Allocates no memory.

@BlockFree
@SCJAllowed
public static double **floor**(double a)

Allocates no memory.

@BlockFree
@SCJAllowed
public static int **getExponent**(float a)

Allocates no memory.

@BlockFree
@SCJAllowed
public static int **getExponent**(double a)

Allocates no memory.

@BlockFree
@SCJAllowed
public static double **hypot**(double x , double y)

Allocates no memory.

@BlockFree
@SCJAllowed
public static double **log**(double a)

Allocates no memory.

@BlockFree
@SCJAllowed
public static double **log10**(double a)

Allocates no memory.

@BlockFree
@SCJAllowed
public static double **log1p**(double a)

Allocates no memory.

@BlockFree
@SCJAllowed
public static double **max**(double a , double b)

Allocates no memory.

@BlockFree
@SCJAllowed
public static long **max**(long a , long b)

Allocates no memory.

@BlockFree
@SCJAllowed
public static float **max**(float a , float b)

Allocates no memory.

@BlockFree
@SCJAllowed
public static int **max**(int a , int b)

Allocates no memory.

@BlockFree
@SCJAllowed
public static long **min**(long a , long b)

Allocates no memory.

@BlockFree
@SCJAllowed
public static double **min**(double a , double b)

Allocates no memory.

@BlockFree
@SCJAllowed
public static float **min**(float a , float b)

Allocates no memory.

@BlockFree
@SCJAllowed
public static int **min**(int a , int b)

Allocates no memory.

@BlockFree
@SCJAllowed
public static float **nextAfter**(float start , float direction)

Allocates no memory.

@BlockFree
@SCJAllowed
public static double **nextAfter**(double start , double direction)

Allocates no memory.

@BlockFree
@SCJAllowed
public static float **nextUp**(float d)

Allocates no memory.

@BlockFree
@SCJAllowed
public static double **nextUp**(double d)

Allocates no memory.

@BlockFree
@SCJAllowed
public static double **pow**(double a , double b)

Allocates no memory.

@BlockFree
@SCJAllowed
public static double **random**( )

Allocates no memory.

@BlockFree
@SCJAllowed
public static double **rint**(double a)

Allocates no memory.

@BlockFree
@SCJAllowed
public static long **round**(double a)

Allocates no memory.

@BlockFree
@SCJAllowed
public static int **round**(float a)

Allocates no memory.

@BlockFree
@SCJAllowed
public static float **scalb**(float f , int scaleFactor)

Allocates no memory.

@BlockFree
@SCJAllowed
public static double **scalb**(double d , int scaleFactor)

Allocates no memory.

@BlockFree
@SCJAllowed
public static float **signum**(float f)

Allocates no memory.

@BlockFree
@SCJAllowed
public static double **signum**(double d)

Allocates no memory.

@BlockFree
@SCJAllowed
public static double **sin**(double a)

Allocates no memory.

@BlockFree
@SCJAllowed
public static double **sinh**(double a)

Allocates no memory.

@BlockFree
@SCJAllowed
public static double **sqrt**(double a)

Allocates no memory.

@BlockFree
@SCJAllowed
public static double **tan**(double a)

Allocates no memory.

@BlockFree
@SCJAllowed
public static double **tanh**(double a)

Allocates no memory.

@BlockFree
@SCJAllowed
public static double **toDegrees**(double val)

Allocates no memory.

@BlockFree
@SCJAllowed
public static double **toRadians**(double val)

Allocates no memory.

@BlockFree
@SCJAllowed
public static float **ulp**(float d)

Allocates no memory.

@BlockFree
@SCJAllowed
public static double **ulp**(double d)

Allocates no memory.

B.2.44 CLASS **String**

DECLARATION

```
@SCJAllowed
public final class String
  implements java.lang.CharSequence, java.lang.Comparable, java.io.Serializable
  extends java.lang.Object
```

CONSTRUCTORS

@BlockFree
@SCJAllowed
public **String**( )

Does not allow ”this” argument to escape local variables.

```
@Allocate({javax.safetycritical.annotate.Allocate.Area.THIS})
@BlockFree
@SCJAllowed
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”.

```
@Allocate({javax.safetycritical.annotate.Allocate.Area.THIS})
@BlockFree
@SCJAllowed
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”.

```
@Allocate({javax.safetycritical.annotate.Allocate.Area.THIS})
@BlockFree
@SCJAllowed
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”.

```
@Allocate({javax.safetycritical.annotate.Allocate.Area.THIS})
@BlockFree
@SCJAllowed
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”.

```
@BlockFree
@MemoryAreaEncloses(inner = {"this"}, outer = {"b"})
@SCJAllowed
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.

```
@Allocate({javax.safetycritical.annotate.Allocate.Area.THIS})
@BlockFree
@SCJAllowed
public String( char []c , int offset , int length )
```

Does not allow "this" or "c" argument to escape local variables. Allocates internal structure to hold the contents of c within the same scope as "this".

METHODS

```
@BlockFree
@SCJAllowed
public char  charAt( int index )
```

Allocates no memory. Does not allow "this" to escape local variables.

```
@BlockFree
@SCJAllowed
public int  compareTo( String str )
```

Allocates no memory. Does not allow "this" or "str" argument to escape local variables.

@BlockFree
@SCJAllowed
public int **compareToIgnoreCase**(String str)

Allocates no memory. Does not allow ”this” or ”str” argument to escape local variables.

@Allocate({javax.safetycritical.annotate.Allocate.Area.CURRENT})
@BlockFree
@SCJAllowed
public 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.

@BlockFree
@SCJAllowed
public boolean **contains**(CharSequence arg)

Does not allow ”this” or ”str” argument to escape local variables.

@BlockFree
@SCJAllowed
public boolean **contentEquals**(CharSequence cs)

Does not allow ”this” or ”str” argument to escape local variables.

@BlockFree
@SCJAllowed
public final boolean **endsWith**(String suffix)

Allocates no memory. Does not allow ”this” or ”suffix” argument to escape local variables.

```
@BlockFree
@SCJAllowed
public boolean  equals( Object obj )
```

Allocates no memory. Does not allow ”this” or ”obj” argument to escape local variables.

```
@BlockFree
@SCJAllowed
public boolean  equalsIgnoreCase( String str )
```

Allocates no memory. Does not allow ”this” or ”str” argument to escape local variables.

```
@Allocate({javax.safetycritical.annotate.Allocate.Area.CURRENT})
@BlockFree
@SCJAllowed
public byte []  getBytes( )
```

Does not allow ”this” to escape local variables. Allocates a byte array in the caller’s context.

```
@BlockFree
@SCJAllowed
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.

```
@BlockFree
@SCJAllowed
public int  hashCode( )
```

Allocates no memory. Does not allow ”this” to escape local variables.

@BlockFree
@SCJAllowed
public int **indexOf**(int ch , int from_index)

Allocates no memory. Does not allow ”this” to escape local variables.

@BlockFree
@SCJAllowed
public int **indexOf**(String str , int from_index)

Allocates no memory. Does not allow ”this” or ”str” argument to escape local variables.

@BlockFree
@SCJAllowed
public int **indexOf**(String str)

Allocates no memory. Does not allow ”this” or ”str” argument to escape local variables.

@BlockFree
@SCJAllowed
public int **indexOf**(int ch)

Allocates no memory. Does not allow ”this” to escape local variables.

@BlockFree
@SCJAllowed
public boolean **isEmpty**( )

Allocates no memory. Does not allow ”this” argument to escape local variables.

@BlockFree
@SCJAllowed
public int **lastIndexOf**(String str)

Allocates no memory. Does not allow ”this” or ”str” argument to escape local variables.

@BlockFree
@SCJAllowed
public int **lastIndexOf**(String str , int from_index)

Allocates no memory. Does not allow ”this” or ”str” argument to escape local variables.

@BlockFree
@SCJAllowed
public int **lastIndexOf**(int ch , int from_index)

Allocates no memory. Does not allow ”this” to escape local variables.

@BlockFree
@SCJAllowed
public int **lastIndexOf**(int ch)

Allocates no memory. Does not allow ”this” to escape local variables.

@BlockFree
@SCJAllowed
public int **length**( )

Allocates no memory. Does not allow ”this” to escape local variables.

@BlockFree
@SCJAllowed
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.

@BlockFree
@SCJAllowed
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.

@Allocate({javax.safetycritical.annotate.Allocate.Area.CURRENT})
@BlockFree
@SCJAllowed
public 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.

@Allocate({javax.safetycritical.annotate.Allocate.Area.CURRENT})
@BlockFree
@SCJAllowed
public 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.

@BlockFree
@SCJAllowed
public final boolean **startsWith**(String prefix , int toffset)

Allocates no memory. Does not allow ”this” or ”prefix” argument to escape local variables.

@BlockFree
@SCJAllowed
public final boolean **startsWith**(String prefix)

Allocates no memory. Does not allow ”this” or ”prefix” argument to escape local variables.

@Allocate({javax.safetycritical.annotate.Allocate.Area.CURRENT})
@BlockFree
@MemoryAreaEncloses(inner = {”@result”}, outer = {”this”})
@SCJAllowed
public 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.

@Allocate({javax.safetycritical.annotate.Allocate.Area.CURRENT})
@BlockFree
@MemoryAreaEncloses(inner = {”@result”}, outer = {”this”})
@SCJAllowed
public 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.

@Allocate({javax.safetycritical.annotate.Allocate.Area.CURRENT})
@BlockFree
@MemoryAreaEncloses(inner = {”@result”}, outer = {”this”})
@SCJAllowed
public 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.

```
@Allocate({javax.safetycritical.annotate.Allocate.Area.CURRENT})
@BlockFree
@SCJAllowed
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.

```
@Allocate({javax.safetycritical.annotate.Allocate.Area.CURRENT})
@BlockFree
@SCJAllowed
public 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.

```
@Allocate({javax.safetycritical.annotate.Allocate.Area.CURRENT})
@BlockFree
@SCJAllowed
public 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()).

```
@Allocate({javax.safetycritical.annotate.Allocate.Area.CURRENT})
@BlockFree
@SCJAllowed
public 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.

```
@Allocate({javax.safetycritical.annotate.Allocate.Area.CURRENT})
@BlockFree
@MemoryAreaEncloses(inner = {"@result"}, outer = {"this"})
@SCJAllowed
public final 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.

```
@Allocate({javax.safetycritical.annotate.Allocate.Area.CURRENT})
@BlockFree
@SCJAllowed
public static String valueOf( float f )
```

Allocates a String and associated internal "structure" (e.g. char[]) in caller's scope.

```
@Allocate({javax.safetycritical.annotate.Allocate.Area.CURRENT})
@BlockFree
@SCJAllowed
public static String valueOf( int i )
```

Allocates a String and associated internal "structure" (e.g. char[]) in caller's scope.

```
@Allocate({javax.safetycritical.annotate.Allocate.Area.CURRENT})
@BlockFree
@SCJAllowed
public static String valueOf( long l )
```

Allocates a String and associated internal "structure" (e.g. char[]) in caller's scope.

```
@Allocate({javax.safetycritical.annotate.Allocate.Area.CURRENT})
@BlockFree
@SCJAllowed
public static String  valueOf( Object o )
```

Allocates a String object in the caller's scope.

```
@Allocate({javax.safetycritical.annotate.Allocate.Area.CURRENT})
@BlockFree
@SCJAllowed
public static 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.

```
@Allocate({javax.safetycritical.annotate.Allocate.Area.CURRENT})
@BlockFree
@SCJAllowed
public static 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.

```
@Allocate({javax.safetycritical.annotate.Allocate.Area.CURRENT})
@BlockFree
@SCJAllowed
public static String  valueOf( double d )
```

Allocates a String and associated internal "structure" (e.g. char[]) in caller's scope.

```
@Allocate({javax.safetycritical.annotate.Allocate.Area.CURRENT})
@BlockFree
@SCJAllowed
public static String  valueOf( char c )
```

Allocates a String and associated internal ”structure” (e.g. char[]) in caller’s scope.

```
@BlockFree
@SCJAllowed
public static String  valueOf( boolean b )
```

Allocates no memory. Returns a preallocated String residing in the scope of the String class’s ClassLoader.

B.2.45 CLASS StringBuilder

DECLARATION

```
@SCJAllowed
public final class StringBuilder
  implements java.lang.Appendable, java.lang.CharSequence, java.io.Serializable
  extends java.lang.Object
```

CONSTRUCTORS

```
@Allocate({javax.safetycritical.annotate.Allocate.Area.THIS})
@BlockFree
@SCJAllowed
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”.

```
@Allocate({javax.safetycritical.annotate.Allocate.Area.THIS})
@BlockFree
@SCJAllowed
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”.

```
@Allocate({javax.safetycritical.annotate.Allocate.Area.THIS})
@BlockFree
@SCJAllowed
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”.

```
@Allocate({javax.safetycritical.annotate.Allocate.Area.THIS})
@BlockFree
@SCJAllowed
public StringBuilder( CharSequence seq )
```

Does not allow ”this” to escape local variables. Allocates a character internal structure of sufficient size to represent seq.length() + 16 characters within the scope of ”this”.

METHODS

```
@Allocate({javax.safetycritical.annotate.Allocate.Area.THIS})
@BlockFree
@SCJAllowed
public 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.

```
@Allocate({javax.safetycritical.annotate.Allocate.Area.THIS})
@BlockFree
@SCJAllowed
public 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.

```
@Allocate({javax.safetycritical.annotate.Allocate.Area.THIS})
@BlockFree
@SCJAllowed
public 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.

```
@Allocate({javax.safetycritical.annotate.Allocate.Area.THIS})
@BlockFree
@SCJAllowed
public 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.

```
@Allocate({javax.safetycritical.annotate.Allocate.Area.THIS})
@BlockFree
@SCJAllowed
public 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.

```
@Allocate({javax.safetycritical.annotate.Allocate.Area.THIS})
@BlockFree
```

@SCJAllowed
public 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.

@Allocate({javax.safetycritical.annotate.Allocate.Area.THIS})
@BlockFree
@SCJAllowed
public StringBuilder **append**(Object o)

Does not allow ”this” to escape local variables. If expansion of ”this” StringBuilder’s internal character buffer is necessary, a new char array is allocated within the scope of ”this”. The new array will be twice the length of the existing array, plus 1.

Requires that argument ”o” reside in a scope that encloses ”this”

@Allocate({javax.safetycritical.annotate.Allocate.Area.THIS})
@BlockFree
@SCJAllowed
public StringBuilder **append**(int i)

Does not allow ”this” to escape local variables. If expansion of ”this” StringBuilder’s internal character buffer is necessary, a new char array is allocated within the scope of ”this”. The new array will be twice the length of the existing array, plus 1.

@Allocate({javax.safetycritical.annotate.Allocate.Area.THIS})
@BlockFree
@SCJAllowed
public StringBuilder **append**(double d)

Does not allow ”this” to escape local variables. If expansion of ”this” StringBuilder’s internal character buffer is necessary, a new char array is allocated within the scope of ”this”. The new array will be twice the length of the existing array, plus 1.

```
@Allocate({javax.safetycritical.annotate.Allocate.Area.THIS})
@BlockFree
@SCJAllowed
public 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.

```
@Allocate({javax.safetycritical.annotate.Allocate.Area.THIS})
@BlockFree
@SCJAllowed
public 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.

```
@Allocate({javax.safetycritical.annotate.Allocate.Area.THIS})
@BlockFree
@SCJAllowed
public StringBuilder append( boolean b )
```

Does not allow ”this” to escape local variables. If expansion of ”this” StringBuilder’s internal character buffer is necessary, a new char array is allocated within the scope of ”this”. The new array will be twice the length of the existing array, plus 1.

```
@BlockFree
@SCJAllowed
```

public int **capacity**( )

Allocates no memory. Does not allow ”this” to escape local variables.

@BlockFree
@SCJAllowed
public char **charAt**(int index)

Allocates no memory. Does not allow ”this” to escape local variables.

@Allocate({javax.safetycritical.annotate.Allocate.Area.THIS})
@BlockFree
@SCJAllowed
public void **ensureCapacity**(int minimum_capacity)

Does not allow ”this” to escape local variables. If expansion of ”this” StringBuilder’s internal character buffer is necessary, a new char array is allocated within the scope of ”this”. The new array will be twice the length of the existing array, plus 1.

@BlockFree
@SCJAllowed
public void **getChars**(int srcBegin , int srcEnd , char []dst , int dstBegin)

Does not allow ”this” or ”dst” to escape local variables.

@BlockFree
@SCJAllowed
public void **indexOf**(String str , int fromIndex)

Does not allow ”this” or ”dst” to escape local variables.

@BlockFree
@SCJAllowed
public void **indexOf**(String str)

Does not allow ”this” or ”dst” to escape local variables.

@BlockFree
@SCJAllowed
public void **lastIndexOf**(String str , int fromIndex)

Does not allow ”this” or ”dst” to escape local variables.

@BlockFree
@SCJAllowed
public void **lastIndexOf**(String str)

Does not allow ”this” or ”dst” to escape local variables.

@Allocate({javax.safetycritical.annotate.Allocate.Area.THIS})
@BlockFree
@SCJAllowed
public int **length**( )

Allocates no memory. Does not allow ”this” to escape local variables.

@BlockFree
@SCJAllowed
public void **setLength**(int new_length)

Does not allow ”this” to escape local variables.

@Allocate({javax.safetycritical.annotate.Allocate.Area.CURRENT})
@BlockFree
@SCJAllowed
public CharSequence **subSequence**(int start , int end)

Does not allow ”this” to escape local variables. Allocates a String in caller’s scope.

```
@Allocate({javax.safetycritical.annotate.Allocate.Area.CURRENT})
@BlockFree
@SCJAllowed
public String  toString( )
```

Does not allow ”this” to escape local variables. Allocates a String in caller’s scope.

B.2.46 CLASS StringIndexOutOfBoundsException

DECLARATION

```
@SCJAllowed
public class StringIndexOutOfBoundsException
  implements java.io.Serializable
  extends java.lang.IndexOutOfBoundsException
```

CONSTRUCTORS

```
@Allocate({javax.safetycritical.annotate.Allocate.Area.CURRENT})
@BlockFree
@SCJAllowed
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).

```
@Allocate({javax.safetycritical.annotate.Allocate.Area.CURRENT})
@BlockFree
@SCJAllowed
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).

```
@Allocate({javax.safetycritical.annotate.Allocate.Area.CURRENT})
@BlockFree
@MemoryAreaEncloses(inner = {”this”}, outer = {”msg”})
@SCJAllowed
public StringIndexOutOfBoundsException( String msg )
```

Shall not copy ”this” to any instance or static field. The scope containing the msg argument must enclose the scope containing ”this”. Otherwise, an IllegalAssignmentError will be thrown.

Allocates an application- and implementation-dependent amount of memory in the current scope (to represent stack backtrace).

B.2.47 CLASS System

DECLARATION

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

CONSTRUCTORS

```
@BlockFree
@SCJAllowed
protected System( )
```

Allocates no memory.

METHODS

```
@BlockFree
@SCJAllowed
```

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. <b\> Requires that the contents of array src enclose array dest. TBD: our annotation system doesn't have a way to describe this scope constraint.

@BlockFree
@SCJAllowed
public static long **currentTimeMillis**( )

Allocates no memory.

@BlockFree
@SCJAllowed
public static void **exit**(int code)

Allocates no memory.

@BlockFree
@SCJAllowed
public static 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.

@BlockFree
@SCJAllowed

public static 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.

@BlockFree
@SCJAllowed
public static int **identityHashCode**(Object x)

Does not allow argument ”x” to escape local variables. Allocates no memory.

@BlockFree
@SCJAllowed
public static long **nanoTime**( )

Allocates no memory.

B.2.48 CLASS **Thread**

DECLARATION

```
@SCJAllowed(javax.safetycritical.annotate.Level.LEVEL_0)
public class Thread
  implements java.lang.Runnable
  extends java.lang.Object
```

METHODS

@BlockFree
@SCJAllowed(javax.safetycritical.annotate.Level.LEVEL_2)

public static Thread.UncaughtExceptionHandler **getDefaultUncaughtExceptionHandler**( )

@memory
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”.

@BlockFree
@SCJAllowed(javax.safetycritical.annotate.Level.LEVEL_2)
public static Thread.UncaughtExceptionHandler **getUncaughtExceptionHandler**( )

@memory
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”.

@BlockFree
@SCJAllowed(javax.safetycritical.annotate.Level.LEVEL_2)
public void **interrupt**( )

@memory
Allocates no memory. Does not allow ”this” to escape local variables.

@BlockFree
@SCJAllowed(javax.safetycritical.annotate.Level.LEVEL_2)
public static boolean **interrupted**( )

@memory
Allocates no memory. Does not allow ”this” to escape local variables.

@BlockFree
@SCJAllowed(javax.safetycritical.annotate.Level.LEVEL_2)
public final boolean **isAlive**( )

@memory
Allocates no memory. Does not allow ”this” to escape local variables.

@BlockFree
@SCJAllowed(javax.safetycritical.annotate.Level.LEVEL_2)

public boolean **isDaemon**( )

@memory
Allocates no memory. Does not allow ”this” to escape local variables.

@BlockFree
@SCJAllowed(javax.safetycritical.annotate.Level.LEVEL_2)
public boolean **isInterrupted**( )

@memory
Allocates no memory. Does not allow ”this” to escape local variables.

@SCJAllowed(javax.safetycritical.annotate.Level.LEVEL_2)
public final void **join**(long millis)

@memory
Allocates no memory. Does not allow ”this” to escape local variables.

@SCJAllowed(javax.safetycritical.annotate.Level.LEVEL_2)
public final void **join**(long millis , int nanos)

@memory
Allocates no memory. Does not allow ”this” to escape local variables.

@SCJAllowed(javax.safetycritical.annotate.Level.LEVEL_2)
public final void **join**( )

@memory
Allocates no memory. Does not allow ”this” to escape local variables.

@BlockFree
@MemoryAreaEncloses(inner = {”@immortal”}, outer = {”eh”})
@SCJAllowed(javax.safetycritical.annotate.Level.LEVEL_2)
public static void **setDefaultUncaughtExceptionHandler**(Thread.UncaughtExceptionHandler eh)

@memory
Allocates no memory. Does not allow ”this” to escape local variables. The eh argument must reside in immortal memory.

@BlockFree

@MemoryAreaEncloses(inner = {"this"}, outer = {"eh"})
@SCJAllowed(javax.safetycritical.annotate.Level.LEVEL_2)
public void **setUncaughtExceptionHandler**(Thread.UncaughtExceptionHandler eh
)

@memory
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".

@Allocate({javax.safetycritical.annotate.Allocate.Area.CURRENT})
@BlockFree
@SCJAllowed
public String **toString**( )

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

@BlockFree
@SCJAllowed(javax.safetycritical.annotate.Level.LEVEL_2)
public static void **yield**( )

@memory
Allocates no memory.

B.2.49 CLASS **Throwable**

DECLARATION

```
@SCJAllowed
public class Throwable
  implements java.io.Serializable
  extends java.lang.Object
```

CONSTRUCTORS

@SCJAllowed
public **Throwable**( )

Shall not copy ”this” to any instance or static field.

Invokes System.captureStackBacktrace(this) to save the back trace associated with the current thread.

```
@BlockFree
@MemoryAreaEncloses(inner = {”this”}, outer = {”cause”})
@SCJAllowed
public Throwable( Throwable cause )
```

Shall not copy ”this” to any instance or static field.

Does not invoke System.captureStackBacktrace(this) so as to not overwrite the backtrace associated with cause.

```
@BlockFree
@MemoryAreaEncloses(inner = {”this”, ”this”}, outer = {”cause”, ”msg”})
@SCJAllowed
public Throwable( String msg , Throwable cause )
```

Shall not copy ”this” to any instance or static field.

Does not invoke System.captureStackBacktrace(this) so as to not overwrite the backtrace associated with cause.

```
@BlockFree
@MemoryAreaEncloses(inner = {”this”}, outer = {”msg”})
@SCJAllowed
public Throwable( String msg )
```

Shall not copy ”this” to any instance or static field. The scope containing the msg argument must enclose the scope containing ”this”. Otherwise, an IllegalAssignmentError will be thrown.

Invokes System.captureStackBacktrace(this) to save the back trace associated with the current thread.

METHODS

@BlockFree
@SCJAllowed
public Throwable **getCause**( )

Performs no memory allocation. 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.

@BlockFree
@SCJAllowed
public String **getMessage**( )

Performs no memory allocation. 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.

@Allocate({javax.safetycritical.annotate.Allocate.Area.CURRENT})
@BlockFree
@SCJAllowed
public StackTraceElement [] **getStackTrace**( )

Shall not copy "this" to any instance or static field.

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.

B.2.50 CLASS **UnsatisfiedLinkError**

DECLARATION

```
@SCJAllowed
public class UnsatisfiedLinkError
  implements java.io.Serializable
  extends java.lang.RuntimeException
```

CONSTRUCTORS

```
@Allocate({javax.safetycritical.annotate.Allocate.Area.CURRENT})
@BlockFree
@SCJAllowed
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).

```
@Allocate({javax.safetycritical.annotate.Allocate.Area.CURRENT})
@BlockFree
@MemoryAreaEncloses(inner = {”this”}, outer = {”msg”})
@SCJAllowed
public UnsatisfiedLinkError( String msg )
```

Shall not copy ”this” to any instance or static field. The scope containing the msg argument must enclose the scope containing ”this”. Otherwise, an IllegalAssignmentError will be thrown.

Allocates an application- and implementation-dependent amount of memory in the current scope (to represent stack backtrace).

B.2.51 CLASS **UnsupportedOperationException**

DECLARATION

```
@SCJAllowed
public class UnsupportedOperationException
  implements java.io.Serializable
  extends java.lang.RuntimeException
```

CONSTRUCTORS

```
@Allocate({javax.safetycritical.annotate.Allocate.Area.CURRENT})
@BlockFree
@SCJAllowed
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).

```
@Allocate({javax.safetycritical.annotate.Allocate.Area.CURRENT})
@BlockFree
@MemoryAreaEncloses(inner = {”this”}, outer = {”msg”})
@SCJAllowed
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).

```
@Allocate({javax.safetycritical.annotate.Allocate.Area.CURRENT})
@BlockFree
@MemoryAreaEncloses(inner = {”this”, ”this”}, outer = {”msg”, ”t”})
@SCJAllowed
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).

```
@Allocate({javax.safetycritical.annotate.Allocate.Area.CURRENT})
@BlockFree
@MemoryAreaEncloses(inner = {”this”}, outer = {”t”})
@SCJAllowed
public UnsupportedOperationException( Throwable t )
```

Shall not copy ”this” to any instance or static field. The scope containing the msg argument must enclose the scope containing ”this”. Otherwise, an IllegalAssignmentError will be thrown.

Allocates an application- and implementation-dependent amount of memory in the current scope (to represent stack backtrace).

B.2.52 CLASS VirtualMachineError

DECLARATION

```
@SCJAllowed
public class VirtualMachineError
  implements java.io.Serializable
  extends java.lang.Error
```

CONSTRUCTORS

```
@Allocate({javax.safetycritical.annotate.Allocate.Area.CURRENT})
@BlockFree
@SCJAllowed
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).

```
@Allocate({javax.safetycritical.annotate.Allocate.Area.CURRENT})
@BlockFree
@MemoryAreaEncloses(inner = {"this"}, outer = {"msg"})
@SCJAllowed
public VirtualMachineError( String msg )
```

Shall not copy "this" to any instance or static field. The scope containing the msg argument must enclose the scope containing "this". Otherwise, an IllegalAssignmentError will be thrown.

Allocates an application- and implementation-dependent amount of memory in the current scope (to represent stack backtrace).

B.2.53 CLASS Void

DECLARATION

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

FIELDS

```
 @SCJAllowed
public static final Class TYPE
```

Appendix C

Javadoc Description of Package javax.microedition.io

Package Contents *Page*

Interfaces

Connection 369

A generic connection that just provides the ability to be closed.

InputConnection 369

A marker for connections that can input data.

OutputConnection 370

A marker for connections that can output data.

StreamConnection 370

A Marker for Connections that can both read and write data.

Classes

ConnectionNotFoundException 370

An exception to throw when the connection for a given URL cannot be created because the resources are not available or no factory exists.

Connector 371

...no description...

C.1 Interfaces

C.1.1 INTERFACE Connection

A generic connection that just provides the ability to be closed.

DECLARATION

```
@SCJAllowed
public interface Connection
```

METHODS

```
@SCJAllowed
public void close( )
```

Clean up all resources for this connection and make it unusable.

C.1.2 INTERFACE InputConnection

A marker for connections that can input data.

DECLARATION

```
@SCJAllowed
public interface InputConnection
  implements javax.microedition.io.Connection
```

METHODS

```
@SCJAllowed
public InputStream openInputStream( )
```

The method for getting a stream from the connection to input data.

C.1.3 INTERFACE **OutputConnection**

A marker for connections that can output data.

DECLARATION

```
@SCJAllowed
public interface OutputConnection
  implements javax.microedition.io.Connection
```

METHODS

```
@SCJAllowed
public OutputStream  openOutputStream( )
```

The method for getting a stream from a connection to output data.

C.1.4 INTERFACE **StreamConnection**

A Marker for Connections that can both read and write data.

DECLARATION

```
@SCJAllowed
public interface StreamConnection
  implements javax.microedition.io.InputConnection,
javax.microedition.io.OutputConnection
```

C.2 Classes

C.2.1 CLASS **ConnectionNotFoundException**

An exception to throw when the connection for a given URL cannot be created because the resources are not available or no factory exists.

DECLARATION

```
@SCJAllowed
public class ConnectionNotFoundException
  extends java.lang.Exception
```

CONSTRUCTORS

```
@SCJAllowed
public ConnectionNotFoundException( String message )
```

Create this exception with a text description.

```
@SCJAllowed
public ConnectionNotFoundException( )
```

Create this exception with no description.

C.2.2 CLASS Connector

DECLARATION

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

FIELDS

```
 @SCJAllowed
public static final int READ
```

```
 @SCJAllowed
public static final int READ_WRITE
```

@SCJAllowed
public static final int WRITE

METHODS

@SCJAllowed
public static Connection **open**(String name , int mode)

@SCJAllowed
public static Connection **open**(String name)

@SCJAllowed
public static InputStream **openInputStream**(String name)

@SCJAllowed
public static OutputStream **openOutputStream**(String name)

Appendix D

Javadoc Description of Package javax.realtime

Package Contents *Page*

Interfaces

AllocationContext 379

This is the base interface for all memory areas.

ClockCallBack 381

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.

EventExaminer 382

Note:

PhysicalMemoryName 382

...no description...

RawIntegralAccess 382

...no description...

RawIntegralAccessFactory 384

...no description...

RawMemoryName 385

...no description...

RawScalarAccess 385

...no description...

RawScalarAccessFactory 385

...no description...

Schedulable 385

...no description...

ScopedAllocationContext 386

This is the base interface for all scoped memory areas.

Classes

AbsoluteTime 387

An object that represents a specific point in time given by milliseconds plus nanoseconds past some point in time fixed by the clock.

AffinitySet 392

...no description...

AperiodicParameters 394

...no description...

AsyncEvent 394

...no description...

AsyncEventHandler 395

...no description...

AutonomousHappening 395

...no description...

BoundAsyncEventHandler 396

...no description...

Clock 396

A clock marks the passing of time.

ControlledHappening 400

Note:

EventHappening 401

...no description...

Happening 402

...no description...

HighResolutionTime 404

Class HighResolutionTime is the base class for AbsoluteTime, RelativeTime, RationalTime.

IllegalAssignmentError 408

...no description...

ImmortalMemory 408

...no description...

InaccessibleAreaException 409

TBD: do we make this SCJAllowed? It may be that the restrictions put in place for JSR 302 code will guarantee that this exception is never thrown.

InterruptHappening 410

Note: IT IS NOT CLEAR WHICH PACKAGE THIS LIVES IN IF THIS DOES NOT APPEAR I AN RTSJ EXTENSION PACKAGE THEN THIS AND ManagedInterruptHappenings SHOULD BE MERGED.

LTMemory 412

...no description...

MemoryAccessError 413

...no description...

MemoryArea 413

...no description...

MemoryInUseException 416

...no description...

MemoryScopeException 417

...no description...

NoHeapRealtimeThread 417

...no description...

PeriodicParameters 418

...no description...

PhysicalMemoryManager 419

...no description...

PriorityParameters 420

...no description...

PriorityScheduler 420

...no description...

ProcessorAffinityException 421

...no description...

RawMemoryAccess 421

...no description...

RealtimeThread 424

...no description...

RelativeTime 425

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.

ReleaseParameters 429

...no description...

Scheduler 429

...no description...

SchedulingParameters 429

...no description...

ScopedCycleException 430

...no description...

SizeEstimator 430

TBD: we need additional methods to allow SizeEstimation of thread stacks.

ThrowBoundaryError 432

...no description...

D.1 Interfaces

D.1.1 INTERFACE **AllocationContext**

This is the base interface for all memory areas. It is a generalization of the Java Heap to allow for alternate forms of memory management. All memory areas implement this interface.

DECLARATION

```
@SCJAllowed
public interface AllocationContext
```

Author
James J. Hunt, aicas GmbH

METHODS

@SCJAllowed
public void **executeInArea**(Runnable logic)

Execute some logic with this memory area as the default allocation context. The effect on the scope stack is specified in the implementing classes.

logic — is the runnable to execute in this memory area.

@SCJAllowed
public long **memoryConsumed**( )

Get the amount of allocated memory in this memory area.

returns the amount of memory consumed.

@SCJAllowed
public long **memoryRemaining**( )

Get the amount of memory available for allocation in this memory area.

returns the amount of memory remaining.

@SCJAllowed
public Object **newArray**(Class type , int number)

Create a new array of the given type in this memory area. This method may be concurrently used by multiple threads.

type — is 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 — is the number of elements the array should have.
returns the new array of type **type** and size
number .
Throws IllegalArgumentException when **number** is less than zero.

@SCJAllowed
public Object **newInstance**(Class type)

Create a new instance of a class in this memory area using its default constructor.

type — is the class of the object to be created
returns a new instance of the given class.
Throws ExceptionInInitializerError when an unexpected exception has occurred in a static initializer.
Throws IllegalAccessException when the class or initializer is inaccessible under Java access control.
Throws InstantiationException when the specified class object could not be instantiated. Possible causes are the class is an interface, abstract class, or array.
Throws InvocationTargetException when the underlying constructor throws an exception.

@SCJAllowed
public long **size**( )

Get the size of this memory area.

returns the current size of this memory area.

D.1.2 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. The callback is de-registered before a method in it is invoked, and the Clock blocks any attempt by another thread to register another callback while control is in a callback.

DECLARATION

```
@SCJAllowed
public interface ClockCallBack
```

METHODS

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

```
@BlockFree
@SCJAllowed(javax.safetycritical.annotate.Level.LEVEL_1)
public void discontinuity( Clock clock , AbsoluteTime updatedTime )
```

clock experienced a time discontinuity. (It changed its time value other than by ticking.) and 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**

Note:

DECLARATION

```
@SCJAllowed(javax.safetycritical.annotate.Level.LEVEL_2)
public interface EventExaminer
```

METHODS

```
@SCJAllowed(javax.safetycritical.annotate.Level.LEVEL_2)
public Object  visit( AsyncEvent ae )
```

D.1.4 INTERFACE **PhysicalMemoryName**

DECLARATION

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

D.1.5 INTERFACE **RawIntegralAccess**

DECLARATION

```
@SCJAllowed(javax.safetycritical.annotate.Level.LEVEL_1)
public interface RawIntegralAccess
```

METHODS

@SCJAllowed(javax.safetycritical.annotate.Level.LEVEL_1)
public byte **getByte**(long offset)

@SCJAllowed(javax.safetycritical.annotate.Level.LEVEL_1)
public void **getBytes**(long offset , byte []bytes , int low , int number)

@SCJAllowed(javax.safetycritical.annotate.Level.LEVEL_1)
public int **getInt**(long offset)

@SCJAllowed(javax.safetycritical.annotate.Level.LEVEL_1)
public void **getInts**(long offset , int []ints , int low , int number)

@SCJAllowed(javax.safetycritical.annotate.Level.LEVEL_1)
public long **getLong**(long offset)

@SCJAllowed(javax.safetycritical.annotate.Level.LEVEL_1)
public void **getLongs**(long offset , long []longs , int low , int number)

@SCJAllowed(javax.safetycritical.annotate.Level.LEVEL_1)
public short **getShort**(long offset)

@SCJAllowed(javax.safetycritical.annotate.Level.LEVEL_1)
public void **getShorts**(long offset , short []shorts , int low , int number)

@SCJAllowed(javax.safetycritical.annotate.Level.LEVEL_1)
public void **setByte**(long offset , byte value)

@SCJAllowed(javax.safetycritical.annotate.Level.LEVEL_1)
public void **setByte**(long offset , long value)

@SCJAllowed(javax.safetycritical.annotate.Level.LEVEL_1)
public void **setBytes**(long offset , byte []bytes , int low , int number)

@SCJAllowed(javax.safetycritical.annotate.Level.LEVEL_1)
public void **setInt**(long offset , int value)

@SCJAllowed(javax.safetycritical.annotate.Level.LEVEL_1)
public void **setInts**(long offset , int []its , int low , int number)

@SCJAllowed(javax.safetycritical.annotate.Level.LEVEL_1)
public void **setLongs**(long offset , long []longs , int low , int number)

@SCJAllowed(javax.safetycritical.annotate.Level.LEVEL_1)
public void **setShort**(long offset , short value)

@SCJAllowed(javax.safetycritical.annotate.Level.LEVEL_1)
public void **setShorts**(long offset , short []shorts , int low , int number)

D.1.6 INTERFACE **RawIntegralAccessFactory**

DECLARATION

```
@SCJAllowed(javax.safetycritical.annotate.Level.LEVEL_1)
public interface RawIntegralAccessFactory
```

METHODS

@SCJAllowed(javax.safetycritical.annotate.Level.LEVEL_1)
public RawMemoryName **getName**( )

@SCJAllowed(javax.safetycritical.annotate.Level.LEVEL_1)
public RawIntegralAccess **newIntegralAccess**(long base , long size)

D.1.7 INTERFACE **RawMemoryName**

DECLARATION

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

D.1.8 INTERFACE **RawScalarAccess**

DECLARATION

```
@SCJAllowed(javax.safetycritical.annotate.Level.LEVEL_0)
public interface RawScalarAccess
  implements javax.realtime.RawIntegralAccess, javax.realtime.RawRealAccess
```

D.1.9 INTERFACE **RawScalarAccessFactory**

DECLARATION

```
@SCJAllowed(javax.safetycritical.annotate.Level.LEVEL_1)
public interface RawScalarAccessFactory
```

METHODS

```
@SCJAllowed(javax.safetycritical.annotate.Level.LEVEL_1)
public RawMemoryName  getName( )
```

```
@SCJAllowed(javax.safetycritical.annotate.Level.LEVEL_1)
public RawScalarAccess  newRawScalarAccess( long base , long size )
```

D.1.10 INTERFACE **Schedulable**

DECLARATION

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

D.1.11 INTERFACE **ScopedAllocationContext**

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. The exact deallocation semantics depend depend on the implementing class.

DECLARATION

```
@SCJAllowed
public interface ScopedAllocationContext
  implements javax.realtime.AllocationContext
```

Author
James J. Hunt, aicas GmbH, 2010

getMaximumSize, visitScopedChildren

METHODS

```
@SCJAllowed
public Object  getPortal( )
```

Get this memory area's portal object. The portal provides a means of passing information between Schedulable objects in a memory area. Assignment rules are enforced on the value returned by getPortal as if the return value were first stored in an object allocated in the current allocation context, then moved to its final destination.

returns the portal object.
Throws MemoryAccessError when a reference to the portal object cannot be stored in the caller's allocation context; that is, if this is "inner" relative to the current allocation context or not on the caller's scope stack.
Throws IllegalAssignmentError when caller is a Java thread.

```
@SCJAllowed
public void  resize( long size )
```

Change the guarenteed and maximum size of the scoped memory area. The method may only be called when the memory area is empty, i.e., all objects have been reclaimed and no Schedulable object has the area as its allocation context.

size — is the new size for this memory area.
Throws IllegalStateException when the area is not empty.

@SCJAllowed
public void **setPortal**(Object object)

Sets the portal object of the memory area to the given object. The object must have been allocated in this ScopedMemory instance.

object — the new portal object
Throws IllegalThreadStateException when the caller is a Java Thread.
Throws IllegalAssignmentError when the object is not allocated in this scoped memory instance and not null.
Throws InaccessibleAreaException when the caller is a Schedulable object, this memory area is not in the caller's scope stack, and object is not null.

D.2 Classes

D.2.1 CLASS **AbsoluteTime**

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.

DECLARATION

```
@SCJAllowed
public class AbsoluteTime
  extends javax.realtime.HighResolutionTime
```

CONSTRUCTORS

```
@BlockFree
@SCJAllowed
public AbsoluteTime( long millis , int nanos )
```

Construct an AbsoluteTime object with time millisecond and nanosecond components past the real-time clock's Epoch.

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.

```
@BlockFree
@SCJAllowed
public AbsoluteTime( AbsoluteTime time )
```

Make a new AbsoluteTime object from the given AbsoluteTime object.

The — AbsoluteTime object which is the source for the copy.

```
@BlockFree
@SCJAllowed
public AbsoluteTime( long millis , int nanos , Clock clock )
```

Construct an AbsoluteTime object with time millisecond and nanosecond components past the epoch for clock.

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.

@BlockFree
@SCJAllowed
public **AbsoluteTime**(Clock clock)

Equivalent to new AbsoluteTime(0,0,clock).

clock — The clock providing the association for the newly constructed object.

METHODS

@BlockFree
@SCJAllowed
public 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.

@BlockFree
@SCJAllowed
public 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.

```
@Allocate({javax.safetycritical.annotate.Allocate.Area.CURRENT})
@BlockFree
@SCJAllowed
public 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.

```
@Allocate({javax.safetycritical.annotate.Allocate.Area.CURRENT})
@BlockFree
@SCJAllowed
public 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.

```
@BlockFree
@SCJAllowed
public 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.

@BlockFree
@SCJAllowed
public 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.

@Allocate({javax.safetycritical.annotate.Allocate.Area.CURRENT})
@BlockFree
@SCJAllowed
public 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.

@Allocate({javax.safetycritical.annotate.Allocate.Area.CURRENT})
@BlockFree
@SCJAllowed
public 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.

D.2.2 CLASS **AffinitySet**

DECLARATION

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

METHODS

```
@SCJAllowed(javax.safetycritical.annotate.Level.LEVEL_2)
public static AffinitySet  generate( BitSet bitSet )
```

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

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

```
@SCJAllowed(javax.safetycritical.annotate.Level.LEVEL_1)
public static final BitSet  getAvailableProcessors( BitSet dest )
```

```
@SCJAllowed(javax.safetycritical.annotate.Level.LEVEL_1)
public static final BitSet  getAvailableProcessors( )
```

@SCJAllowed(javax.safetycritical.annotate.Level.LEVEL_1)
public final BitSet **getBitSet**( )

@SCJAllowed(javax.safetycritical.annotate.Level.LEVEL_1)
public static final AffinitySet **getNoHeapSoDefaultAffinity**( )

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

@SCJAllowed(javax.safetycritical.annotate.Level.LEVEL_1)
public static AffinitySet [] **getPredefinedAffinitySets**(AffinitySet []dest)

@SCJAllowed(javax.safetycritical.annotate.Level.LEVEL_1)
public static AffinitySet [] **getPredefinedAffinitySets**( )

@SCJAllowed(javax.safetycritical.annotate.Level.LEVEL_2)
public final BitSet **getProcessors**(BitSet dest)

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

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

@SCJAllowed(javax.safetycritical.annotate.Level.LEVEL_0)
public static final void **setProcessorAffinity**(AffinitySet set , BoundAsyncEventHandler aeh)

D.2.3 CLASS **AperiodicParameters**

DECLARATION

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

CONSTRUCTORS

@SCJAllowed(javax.safetycritical.annotate.Level.LEVEL_1)
public **AperiodicParameters**( )

D.2.4 CLASS **AsyncEvent**

DECLARATION

```
@SCJAllowed(javax.safetycritical.annotate.Level.LEVEL_1)
public class AsyncEvent
  extends java.lang.Object
```

METHODS

@SCJAllowed(javax.safetycritical.annotate.Level.LEVEL_1)
public void **fire**( )

fire this event, i.e., releases the execution of all handlers that were added to this event.

@memory
Does not allocate memory. Does not allow this to escape local variables.

D.2.5 CLASS **AsyncEventHandler**

DECLARATION

```
@SCJAllowed(javax.safetycritical.annotate.Level.LEVEL_0)
public class AsyncEventHandler
  implements javax.realtime.Schedulable
  extends java.lang.Object
```

D.2.6 CLASS **AutonomousHappening**

DECLARATION

```
@SCJAllowed(javax.safetycritical.annotate.Level.LEVEL_1)
public class AutonomousHappening
  extends javax.realtime.EventHappening
```

CONSTRUCTORS

@SCJAllowed(javax.safetycritical.annotate.Level.LEVEL_1)
public **AutonomousHappening**( )

Creates a Happening in the current memory area with a system assigned name and id.

@SCJAllowed(javax.safetycritical.annotate.Level.LEVEL_1)
public **AutonomousHappening**(int id)

Creates a Happening in the current memory area with the specified id and a system-assigned name.

@SCJAllowed(javax.safetycritical.annotate.Level.LEVEL_1)
public **AutonomousHappening**(int id , String name)

Creates a Happening in the current memory area with the name and id given.

```
@SCJAllowed(javax.safetycritical.annotate.Level.LEVEL_1)
public AutonomousHappening( String name )
```

Creates a Happening in the current memory area with the name name and a system-assigned id.

D.2.7 CLASS BoundAsyncEventHandler

DECLARATION

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

D.2.8 CLASS Clock

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.

TBD: is the following still true for us? I (MS) assume that Kelvin would like to drive scheduling with user defined clocks.

HighResolutionTime instances that use other clocks are not valid for any purpose that involves sleeping or waiting, including in members of the RealtimeThread.waitForNextPeriod() family. They may, however, be used in the fire time and the period of OneShotTimer and PeriodicTimer.

DECLARATION

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

CONSTRUCTORS

@Allocate({javax.safetycritical.annotate.Allocate.Area.THIS})
@BlockFree
@SCJAllowed
public **Clock**( )

Constructor for the abstract class.

Allocates resolution here.

METHODS

@BlockFree
@SCJAllowed
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() equal 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.

@Allocate({javax.safetycritical.annotate.Allocate.Area.CURRENT})
@BlockFree
@SCJAllowed
public abstract 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.

@BlockFree
@SCJAllowed
public static 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.

@BlockFree
@SCJAllowed
public abstract RelativeTime **getResolution**( )

Gets the resolution of the clock, the nominal interval between ticks.

returns previously allocated resolution object.

@BlockFree
@SCJAllowed
public abstract RelativeTime **getResolution**(RelativeTime dest)

Gets the resolution of the clock, the nominal interval between ticks.

TBD: getTime with a destination null will ignore it and return null. This method (getResolution) will allocated a new object when dest is null.

dest — return the relative time value in dest. If dest is null, allocate a new RelativeTime instance to hold the returned value.
returns dest set to values representing the resolution of this. The returned object is associated with this clock.

@BlockFree
@SCJAllowed
public abstract 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. Note: 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.

@Allocate({javax.safetycritical.annotate.Allocate.Area.CURRENT})
@BlockFree
@SCJAllowed
public abstract AbsoluteTime **getTime**( )

Gets the current time in a newly allocated object. Note: This method will return an absolute time value that represents the clock's notion of an absolute time. For clocks that do not measure calendar time this absolute time may not represent a wall clock time.

returns A newly allocated instance of AbsoluteTime in the current allocation context, representing the current time. The returned object is associated with this clock.

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

This method behaves effectively as if it and invocations of clock events by this clock hold a common 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.

@BlockFree
@SCJAllowed(javax.safetycritical.annotate.Level.LEVEL_1)
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.2.9 CLASS **ControlledHappening**

Note:

DECLARATION

```
@SCJAllowed(javax.safetycritical.annotate.Level.LEVEL_2)
public class ControlledHappening
  extends javax.realtime.EventHappening
```

CONSTRUCTORS

@SCJAllowed(javax.safetycritical.annotate.Level.LEVEL_2)
public **ControlledHappening**( )

@SCJAllowed(javax.safetycritical.annotate.Level.LEVEL_2)
public **ControlledHappening**(int id)

@SCJAllowed(javax.safetycritical.annotate.Level.LEVEL_2)
public **ControlledHappening**(int id , String name)

@SCJAllowed(javax.safetycritical.annotate.Level.LEVEL_2)
public **ControlledHappening**(String name)

METHODS

@SCJAllowed(javax.safetycritical.annotate.Level.LEVEL_2)
public final void **attach**(AsyncEvent ae)

@SCJAllowed(javax.safetycritical.annotate.Level.LEVEL_2)
protected void **process**( )

@SCJAllowed(javax.safetycritical.annotate.Level.LEVEL_2)
public final void **takeControl**( )

@SCJAllowed(javax.safetycritical.annotate.Level.LEVEL_2)
public final void **takeControlInterruptible**( )

@SCJAllowed(javax.safetycritical.annotate.Level.LEVEL_2)
protected final Object **visit**(EventExaminer logic)

D.2.10 CLASS **EventHappening**

DECLARATION

```
@SCJAllowed(javax.safetycritical.annotate.Level.LEVEL_1)
public abstract class EventHappening
  extends javax.realtime.Happening
```

METHODS

@SCJAllowed(javax.safetycritical.annotate.Level.LEVEL_1)
public void **attach**(AsyncEvent ae)

Attach the AsyncEvent ae to this Happening. ADD LEVEL CONSTRAINTS????

Throws ???? if called from outside the mission initialization phase.

@SCJAllowed(javax.safetycritical.annotate.Level.LEVEL_1)
public void **detach**(AsyncEvent ae)

Detach the AsyncEvent ae from this Happening.

Throws ???? if called from outside the mission initialization phase.

D.2.11 CLASS **Happening**

DECLARATION

```
@SCJAllowed(javax.safetycritical.annotate.Level.LEVEL_1)
public abstract class Happening
  extends java.lang.Object
```

METHODS

@SCJAllowed(javax.safetycritical.annotate.Level.LEVEL_1)
public static Happening **getHappening**(String name)

Find a happening by its name.

@SCJAllowed(javax.safetycritical.annotate.Level.LEVEL_1)
public static int **getId**(String name)

Return the ID of the happening with the name name. If there is not happening with that name return 0.

@SCJAllowed(javax.safetycritical.annotate.Level.LEVEL_1)
public final int **getId**( )

Return the id of this happening.

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

Returns the string name of this happening

@SCJAllowed(javax.safetycritical.annotate.Level.LEVEL_1)
public static boolean **isHappening**(String name)

Is there a Happening with name name?

returns True if there is.

@SCJAllowed(javax.safetycritical.annotate.Level.LEVEL_1)
public boolean **isRegistered**( )

returns Return true if this happening is presently registered.

@SCJAllowed(javax.safetycritical.annotate.Level.LEVEL_1)
public final void **register**( )

Register this Happening.

@mem

Throws ???? if called from outside the mission initialization phase.

@SCJAllowed(javax.safetycritical.annotate.Level.LEVEL_1)
public static final boolean **trigger**(int happeningId)

Causes the happening corresponding to happeningId to occur.

returns true if a happening with id happeningId was found, false otherwise.

@SCJAllowed(javax.safetycritical.annotate.Level.LEVEL_1)
public final void **unRegister**( )

Unregister this Happening.

Throws ???? if called from outside the mission initialization phase.

D.2.12 CLASS **HighResolutionTime**

Class HighResolutionTime is the base class for AbsoluteTime, RelativeTime, RationalTime. Used to express time with nanosecond accuracy. This class is never used directly: it is abstract and has no public constructor. Instead, one of its subclasses AbsoluteTime, RelativeTime, or RationalTime should be used.

DECLARATION

```
@SCJAllowed
public abstract class HighResolutionTime
  implements java.lang.Comparable
  extends java.lang.Object
```

METHODS

@BlockFree
@SCJAllowed
public int **compareTo**(HighResolutionTime time)

Compares this HighResolutionTime with the specified HighResolutionTime time.

time — Compares with the time of this.
returns

@BlockFree
@SCJAllowed
public int **compareTo**(Object object)

Compares this HighResolutionTime with the specified object.

object — Compares with the time of this.
returns

@BlockFree
@SCJAllowed
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.

@BlockFree
@SCJAllowed
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.

@BlockFree
@SCJAllowed
public Clock **getClock**( )

returns A reference to the clock associated with this.

@BlockFree
@SCJAllowed
public final long **getMilliseconds**( )

returns The milliseconds component of the time represented by this.

@BlockFree
@SCJAllowed
public final int **getNanoseconds**( )

returns The nanoseconds component of the time represented by this.

@BlockFree
@SCJAllowed
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.

@BlockFree
@SCJAllowed
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.

@BlockFree
@SCJAllowed
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.

@BlockFree
@SCJAllowed
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)

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.
Throws java.lang.InterruptedException java.lang.InterruptedException

D.2.13 CLASS **IllegalAssignmentError**

DECLARATION

```
@SCJAllowed
public class IllegalAssignmentError
  implements java.io.Serializable
  extends java.lang.Error
```

CONSTRUCTORS

```
@Allocate({javax.safetycritical.annotate.Allocate.Area.CURRENT})
@BlockFree
@SCJAllowed
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).

```
@Allocate({javax.safetycritical.annotate.Allocate.Area.CURRENT})
@BlockFree
@MemoryAreaEncloses(inner = {”this”}, outer = {”description”})
@SCJAllowed
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 IllegalAssignmentError will be thrown.

Allocates an application- and implementation-dependent amount of memory in the current scope (to represent stack backtrace).

D.2.14 CLASS **ImmortalMemory**

DECLARATION

```
@SCJAllowed
public final class ImmortalMemory
  extends javax.realtime.MemoryArea
```

METHODS

@BlockFree
@SCJAllowed
public void **enter**(Runnable logic)

@BlockFree
@SCJAllowed
public static ImmortalMemory **instance**( )

@BlockFree
@SCJAllowed
public long **memoryConsumed**( )

@BlockFree
@SCJAllowed
public long **memoryRemaining**( )

@BlockFree
@SCJAllowed
public long **size**( )

D.2.15 CLASS InaccessibleAreaException

TBD: do we make this SCJAllowed? It may be that the restrictions put in place for JSR 302 code will guarantee that this exception is never thrown. However, such restrictions are not yet sufficiently defined to allow this determination.

DECLARATION

```
@SCJAllowed
public class InaccessibleAreaException
  implements java.io.Serializable
  extends java.lang.RuntimeException
```

CONSTRUCTORS

```
@Allocate({javax.safetycritical.annotate.Allocate.Area.CURRENT})
@BlockFree
@SCJAllowed
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).

```
@Allocate({javax.safetycritical.annotate.Allocate.Area.CURRENT})
@BlockFree
@MemoryAreaEncloses(inner = {”this”}, outer = {”msg”})
@SCJAllowed
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).

D.2.16 CLASS InterruptHappening

Note: IT IS NOT CLEAR WHICH PACKAGE THIS LIVES IN IF THIS DOES NOT APPEAR I AN RTSJ EXTENSION PACKAGE THEN THIS AND ManagedInterruptHappenings SHOULD BE MERGED.

DECLARATION

```
@SCJAllowed(javax.safetycritical.annotate.Level.LEVEL_1)
public class InterruptHappening
  extends javax.realtime.Happening
```

CONSTRUCTORS

@SCJAllowed(javax.safetycritical.annotate.Level.LEVEL_1)
public **InterruptHappening**( )

@SCJAllowed(javax.safetycritical.annotate.Level.LEVEL_1)
public **InterruptHappening**(int id)

@SCJAllowed(javax.safetycritical.annotate.Level.LEVEL_1)
public **InterruptHappening**(int id , String name)

@SCJAllowed(javax.safetycritical.annotate.Level.LEVEL_1)
public **InterruptHappening**(String name)

METHODS

@SCJAllowed(javax.safetycritical.annotate.Level.LEVEL_1)
public final int **getPriority**(int id)

@SCJAllowed(javax.safetycritical.annotate.Level.LEVEL_1)
protected void **process**( )

D.2.17 CLASS **LTMemory**

DECLARATION

```
@SCJAllowed
public class LTMemory
  extends javax.realtime.ScopedMemory
```

METHODS

```
@BlockFree
@SCJAllowed(javax.safetycritical.annotate.Level.LEVEL_1)
public void  enter( Runnable logic )
```

> In vanilla RTSJ, enter() is not necessarily block-free because entering an LTMemory region may have to wait for the region to be finalized. However, a compliant implementation of JSR 302 shall provide a block-free implementation of enter. Note that JSR 302 specifies that finalization of LTMemory regions is not performed.

```
@BlockFree
@SCJAllowed
public long  memoryConsumed( )
```

```
@BlockFree
@SCJAllowed
public long  memoryRemaining( )
```

```
@Override
@SCJAllowed
public void  resize( long size )
```

See Also: javax.realtime.ScopedAllocationContext.resize(long)

```
@BlockFree
@SCJAllowed
public long  size( )
```

D.2.18 CLASS **MemoryAccessError**

DECLARATION

```
@SCJAllowed
public class MemoryAccessError
  implements java.io.Serializable
  extends java.lang.RuntimeException
```

CONSTRUCTORS

```
@Allocate({javax.safetycritical.annotate.Allocate.Area.CURRENT})
@BlockFree
@SCJAllowed
public MemoryAccessError( )
```

```
@Allocate({javax.safetycritical.annotate.Allocate.Area.CURRENT})
@BlockFree
@MemoryAreaEncloses(inner = {"this"}, outer = {"msg"})
@SCJAllowed
public MemoryAccessError( String description )
```

D.2.19 CLASS **MemoryArea**

DECLARATION

```
@SCJAllowed
public abstract class MemoryArea
  implements javax.realtime.AllocationContext
  extends java.lang.Object
```

METHODS

```
@BlockFree
@SCJAllowed
public abstract void  enter( Runnable logic )
```

```
@BlockFree
@Allocate(sameAreaAs = {"object"})
@MemoryAreaEncloses(inner = {"logic"}, outer = {"this"})
@SCJAllowed
public void executeInArea( Runnable logic )
```

TBD: This method has no object argument, so this commentary is not meaningful.

Execute **logic** in the memory area containing

object .

"@param" object is the reference for determining the area in which to execute **logic** .

logic — is the runnable to execute in the memory area containing **object** .

```
@BlockFree
@SCJAllowed
public static MemoryArea getMemoryArea( Object object )
```

```
@BlockFree
@SCJAllowed
public abstract long memoryConsumed( )
```

```
@BlockFree
@SCJAllowed
public abstract long memoryRemaining( )
```

```
@Allocate(sameAreaAs = {"this.area"})
@BlockFree
@SCJAllowed
public Object newArray( Class type , int size )
```

This method creates an object of type **type** in the memory area containing **object** .

type — is the type of the object returned.
returns a new object of type **type**

@Allocate(sameAreaAs = {"object"})
@BlockFree
@SCJAllowed
public Object **newArrayInArea**(Object object , Class type , int size)

This method creates an array of type **type** in the memory area containing **object** .

object — is the reference for determining the area in which to allocate the array.
type — is the type of the array element for the returned array.
size — is the size of the array to return.
returns a new array of element type **type** with size
size .

@Allocate(sameAreaAs = {"this.area"})
@BlockFree
@SCJAllowed
public Object **newInstance**(Class type)

TBD: this method has no object argument, so this commentary is not meaningful

This method creates an object of type **type** in the memory area containing **object** .

"@param" object is the reference for determining the area in which to allocate the array.

type — is the type of the object returned.
returns a new object of type **type**
Throws IllegalAccessException IllegalAccessException
Throws IllegalArgumentException IllegalArgumentException
Throws InstantiationException InstantiationException
Throws OutOfMemoryError OutOfMemoryError

Throws ExceptionInInitializerError ExceptionInInitializerError
Throws InaccessibleAreaException InaccessibleAreaException

@Allocate(sameAreaAs = {"object"})
@SCJAllowed
public Object **newInstanceInArea**(Object object , Class type)

This method creates an object of type **type** in the memory area containing **object** .

object — is the reference for determining the area in which to allocate the array.
type — is the type of the object returned.
returns a new object of type **type**

@BlockFree
@SCJAllowed
public abstract long **size**( )

D.2.20 CLASS MemoryInUseException

DECLARATION

```
@SCJAllowed
public class MemoryInUseException
  extends java.lang.RuntimeException
```

CONSTRUCTORS

@BlockFree
@SCJAllowed
public **MemoryInUseException**( )

@Allocate({javax.safetycritical.annotate.Allocate.Area.IMMORTAL})
@BlockFree
@SCJAllowed
public **MemoryInUseException**(String description)

D.2.21 CLASS **MemoryScopeException**

DECLARATION

```
@SCJAllowed
public class MemoryScopeException
  extends java.lang.RuntimeException
```

CONSTRUCTORS

```
@BlockFree
@SCJAllowed
public MemoryScopeException( )
```

```
@Allocate({javax.safetycritical.annotate.Allocate.Area.IMMORTAL})
@BlockFree
@SCJAllowed
public MemoryScopeException( String description )
```

D.2.22 CLASS **NoHeapRealtimeThread**

DECLARATION

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

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)
@BlockFree
public void **start**( )

Creation of thread may block, but starting shall not

D.2.23 CLASS **PeriodicParameters**

DECLARATION

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

CONSTRUCTORS

@MemoryAreaEncloses(inner = {"this", "this"}, outer = {"start", "period"})
@SCJAllowed
@BlockFree
public **PeriodicParameters**(HighResolutionTime start , RelativeTime period)

@memory
Does not allocate memory. Does not allow this to escape local variables. Builds links from this to start and period. Thus, start and period must reside in scopes that enclose this.

TBD: If this maintains references to start and period, then we really should make sure that RelativeTime is immutable. Otherwise, we should make internal copies of these parameters. ****AJW NO – THE COPY IS DONE on creation of schedulable object

METHODS

@BlockFree
@SCJAllowed(javax.safetycritical.annotate.Level.LEVEL_1)
public RelativeTime **getPeriod**( )

returns Returns the object originally passed in to the constructor, which is known to reside in a memory area that encloses this.

@BlockFree
@SCJAllowed
public HighResolutionTime **getStart**( )

returns Returns the object originally passed in to the constructor, which is known to reside in a memory area that encloses this.

D.2.24 CLASS **PhysicalMemoryManager**

DECLARATION

```
@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 PhysicalMemoryName DEVICE

@SCJAllowed(javax.safetycritical.annotate.Level.LEVEL_1)
public static final PhysicalMemoryName DMA

@SCJAllowed(javax.safetycritical.annotate.Level.LEVEL_1)
public static final PhysicalMemoryName IO_PAGE

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

public static final PhysicalMemoryName SHARED

D.2.25 CLASS **PriorityParameters**

DECLARATION

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

CONSTRUCTORS

```
@BlockFree
@SCJAllowed
public PriorityParameters( int priority )
```

METHODS

```
@BlockFree
@SCJAllowed
public int  getPriority( )
```

D.2.26 CLASS **PriorityScheduler**

DECLARATION

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

METHODS

```
@BlockFree
@SCJAllowed
```

public int **getMaxPriority**( )

@BlockFree
@SCJAllowed
public int **getMinPriority**( )

@BlockFree
@SCJAllowed
public int **getNormPriority**( )

D.2.27 CLASS ProcessorAffinityException

DECLARATION

```
@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.2.28 CLASS RawMemoryAccess

DECLARATION

```
@SCJAllowed(javax.safetycritical.annotate.Level.LEVEL_0)
public class RawMemoryAccess
  implements javax.realtime.RawIntegralAccess
  extends java.lang.Object
```

FIELDS

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

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

CONSTRUCTORS

@SCJAllowed(javax.safetycritical.annotate.Level.LEVEL_0)
public **RawMemoryAccess**(PhysicalMemoryName type , long size)

@SCJAllowed(javax.safetycritical.annotate.Level.LEVEL_0)
public **RawMemoryAccess**(PhysicalMemoryName type , long base , long size)

METHODS

@SCJAllowed(javax.safetycritical.annotate.Level.LEVEL_0)
public static RawIntegralAccess **createRmaInstance**(RawMemoryName type , long base , long size)

@SCJAllowed(javax.safetycritical.annotate.Level.LEVEL_0)
public byte **getByte**(long offset)

@SCJAllowed(javax.safetycritical.annotate.Level.LEVEL_0)
public void **getBytes**(long offset , byte []bytes , int low , int number)

@SCJAllowed(javax.safetycritical.annotate.Level.LEVEL_0)
public int **getInt**(long offset)

@SCJAllowed(javax.safetycritical.annotate.Level.LEVEL_0)
public void **getInts**(long offset , int []ints , int low , int number)

@SCJAllowed(javax.safetycritical.annotate.Level.LEVEL_0)
public long **getLong**(long offset)

@SCJAllowed(javax.safetycritical.annotate.Level.LEVEL_0)
public void **getLongs**(long offset , long []longs , int low , int number)

@SCJAllowed(javax.safetycritical.annotate.Level.LEVEL_0)
public short **getShort**(long offset)

@SCJAllowed(javax.safetycritical.annotate.Level.LEVEL_0)
public void **getShorts**(long offset , short []shorts , int low , int number)

@SCJAllowed(javax.safetycritical.annotate.Level.LEVEL_0)
public void **setByte**(long offset , byte value)

@SCJAllowed(javax.safetycritical.annotate.Level.LEVEL_0)
public void **setByte**(long offset , long value)

@SCJAllowed(javax.safetycritical.annotate.Level.LEVEL_0)
public void **setBytes**(long offset , byte []bytes , int low , int number)

@SCJAllowed(javax.safetycritical.annotate.Level.LEVEL_0)
public void **setInt**(long offset , int value)

@SCJAllowed(javax.safetycritical.annotate.Level.LEVEL_0)
public void **setInts**(long offset , int []its , int low , int number)

@SCJAllowed(javax.safetycritical.annotate.Level.LEVEL_0)
public void **setLongs**(long offset , long []longs , int low , int number)

@SCJAllowed(javax.safetycritical.annotate.Level.LEVEL_0)
public void **setShort**(long offset , short value)

@SCJAllowed(javax.safetycritical.annotate.Level.LEVEL_0)
public void **setShorts**(long offset , short []shorts , int low , int number)

D.2.29 CLASS **RealtimeThread**

DECLARATION

```
@SCJAllowed(javax.safetycritical.annotate.Level.LEVEL_1)
public class RealtimeThread
  implements javax.realtime.Schedulable
  extends java.lang.Thread
```

METHODS

@BlockFree
@SCJAllowed(javax.safetycritical.annotate.Level.LEVEL_2)
public static RealtimeThread **currentRealtimeThread**( )

Allocates no memory. Returns an object that resides in the current mission's MissionMemory.

@BlockFree
@SCJAllowed(javax.safetycritical.annotate.Level.LEVEL_1)
public static MemoryArea **getCurrentMemoryArea**( )

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

```
@BlockFree
@SCJAllowed(javax.safetycritical.annotate.Level.LEVEL_2)
public 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.

```
@BlockFree
@SCJAllowed(javax.safetycritical.annotate.Level.LEVEL_2)
public static void sleep( HighResolutionTime time )
```

D.2.30 CLASS **RelativeTime**

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.

DECLARATION

```
@SCJAllowed
public class RelativeTime
  extends javax.realtime.HighResolutionTime
```

CONSTRUCTORS

```
@BlockFree
@SCJAllowed
public RelativeTime( )
```

Equivalent to new RelativeTime(0,0).

@BlockFree
@SCJAllowed
public **RelativeTime**(long ms , int ns)

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.

@BlockFree
@SCJAllowed
public **RelativeTime**(Clock clock)

Equivalent to new RelativeTime(0,0,clock).

clock — The clock providing the association for the newly constructed object.

@BlockFree
@SCJAllowed
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.

@BlockFree
@SCJAllowed
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

```
@Allocate({javax.safetycritical.annotate.Allocate.Area.CURRENT})
@BlockFree
@SCJAllowed
public 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.

```
@BlockFree
@SCJAllowed
public 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.

```
@BlockFree
@SCJAllowed
public 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.

```
@Allocate({javax.safetycritical.annotate.Allocate.Area.CURRENT})
@BlockFree
@SCJAllowed
public 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.

```
@BlockFree
@SCJAllowed
public 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.

```
@Allocate({javax.safetycritical.annotate.Allocate.Area.CURRENT})
@BlockFree
@SCJAllowed
public 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.

D.2.31 CLASS **ReleaseParameters**

DECLARATION

```
@SCJAllowed(javax.safetycritical.annotate.Level.LEVEL_0)
public abstract class ReleaseParameters
  extends java.lang.Object
```

D.2.32 CLASS **Scheduler**

DECLARATION

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

D.2.33 CLASS **SchedulingParameters**

DECLARATION

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

D.2.34 CLASS **ScopedCycleException**

DECLARATION

```
@SCJAllowed
public class ScopedCycleException
  implements java.io.Serializable
  extends java.lang.RuntimeException
```

CONSTRUCTORS

```
@Allocate({javax.safetycritical.annotate.Allocate.Area.CURRENT})
@BlockFree
@SCJAllowed
public ScopedCycleException( )
```

```
@Allocate({javax.safetycritical.annotate.Allocate.Area.CURRENT})
@BlockFree
@MemoryAreaEncloses(inner = {"this"}, outer = {"msg"})
@SCJAllowed
public ScopedCycleException( String description )
```

D.2.35 CLASS **SizeEstimator**

TBD: we need additional methods to allow SizeEstimation of thread stacks. In particular, we need to be able to reserve memory for backing store. Perhaps this belongs in a javax.safetycritical variant of SizeEstimator.

DECLARATION

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

CONSTRUCTORS

@BlockFree
@SCJAllowed
public **SizeEstimator**( )

METHODS

@BlockFree
@SCJAllowed
public long **getEstimate**( )

JSR 302 tightens the semantic requirements on the implementation of getEstimate. For compliance with JSR 302, getEstimate() must return a conservative upper bound on the amount of memory required to represent all of the memory reservations associated with this SizeEstimator object.

@BlockFree
@SCJAllowed
public void **reserve**(SizeEstimator size , int num)

@BlockFree
@SCJAllowed
public void **reserve**(SizeEstimator size)

@BlockFree
@SCJAllowed
public void **reserve**(Class clazz , int num)

@BlockFree
@SCJAllowed
public void **reserveArray**(int length , Class type)

@BlockFree
@SCJAllowed

public void **reserveArray**(int length)

D.2.36 CLASS **ThrowBoundaryError**

DECLARATION

```
@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*

Interfaces

ManagedSchedulable 437

An interface implemented by all Safety Critical Java Schedulable classes.

Safelet 437

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

Schedulable 438

...no description...

Classes

AperiodicEvent 439

TBD(kdn - july 5, 2010): Note that Mission.

AperiodicEventHandler 440

...no description...

Cyclet 441

TBD: Does the JSR302 expert group approve of the following revision?

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

CyclicExecutive 443

TBD: An earlier version of CyclicExecutive extended Mission.

CyclicSchedule 445

A CyclicSchedule represents a time-driven sequence of firings for deterministic scheduling of periodic event handlers.

CyclicSchedule.Frame 446

...no description...

InterruptHandler 447

...no description...

InterruptHappening 448

...no description...

Level0Mission 448

A Level-Zero Safety Critical Java application is comprised of one or more Level0Missions.

Level0MissionSequencer 449

A MissionSequencer runs a sequence of independent Missions interleaved with repeated execution of certain Missions.

ManagedEventHandler 450

...no description...

ManagedInterruptHappening 451

...no description...

ManagedMemory 452

This is the base class for all safety critical Java memory areas.

ManagedThread 453

...no description...

Mission 454

A Safety Critical Java application is comprised of one or more Missions.

MissionSequencer 457

A MissionSequencer runs a sequence of independent Missions interleaved with repeated execution of certain Missions.

NoHeapRealtimeThread 459

...no description...

PeriodicEventHandler 460

...no description...

PortalExtender 462

TBD: what is this?

PriorityScheduler 462

...no description...

PrivateMemory 462

...no description...

Services 463

System wide information

SingleMissionSequencer 465

...no description...

StorageConfigurationParameters 466

...no description...

StorageParameters 467

...no description...

Terminal 469

A simple Terminal that puts out UTF8 version of String/StringBuilder,.

ThrowBoundaryError 470

One ThrowBoundaryError is preallocated for each Schedulable in its outer-most private scope.

E.1 Interfaces

E.1.1 INTERFACE **ManagedSchedulable**

An interface implemented by all Safety Critical Java Schedulable classes. It defines the register mechanism.

DECLARATION

```
@SCJAllowed
public interface ManagedSchedulable
```

METHODS

```
@SCJAllowed
public void register( )
```

Register the task with its Mission.

E.1.2 INTERFACE **Safelet**

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.

Given the implementation s of Safelet that represents a particular SCJ application, the SCJ infrastructure invokes in sequence s.setUp() followed by s.getSequencer(). For the MissionSequencer q returned from s.getSequencer(), the SCJ infrastructure arranges for an independent thread to begin executing the code forthat sequencer and then waits for that thread to terminate its execution. Upon termination of the MissionSequencer's thread, the SCJ infrastructure invokes s.tearDown().

DECLARATION

```
@SCJAllowed
public interface Safelet
```

METHODS

@SCJAllowed
@SCJRestricted({javax.safetycritical.annotate.Restrict.INITIALIZATION})
public MissionSequencer **getSequencer**( )

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

@SCJAllowed
public void **setUp**( )

Code to execute before the sequencer starts.

@SCJAllowed
public void **tearDown**( )

Code to execute after the sequencer ends.

E.1.3 INTERFACE Schedulable

DECLARATION

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

METHODS

@BlockFree
@SCJAllowed(javax.safetycritical.annotate.Level.LEVEL_2)
public 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**

TBD(kdn - july 5, 2010): Note that Mission.requestTermination() must disable all AperiodicEvent objects associated with the Mission, in order to arrange that all AperiodicEventHandlers associated with the Mission can be terminated and joined. This means that the Mission needs to keep track of all AperiodicEvents, so we really need to ”manage” AperiodicEvents, and I believe this means we’ll have to register each one in the Mission.initialize() code. Should this class extend an abstract ManagedEvent class?

DECLARATION

```
@SCJAllowed(javax.safetycritical.annotate.Level.LEVEL_1)
public class AperiodicEvent
  extends javax.realtime.AsyncEvent
```

CONSTRUCTORS

```
@Allocate({javax.safetycritical.annotate.Allocate.Area.THIS})
@MemoryAreaEncloses(inner = {”this”}, outer = {”handler”})
@SCJAllowed(javax.safetycritical.annotate.Level.LEVEL_1)
public AperiodicEvent( AperiodicEventHandler handler )
```

Constructor for an aperiodic event that is linked to a given handler.

Does not allocate memory. Does not allow this to escape the local variables. Builds a link from “this” to handler, so handler must reside in memory that encloses “this”.

handler — – the handler that is to be added to this event.

```
@Allocate({javax.safetycritical.annotate.Allocate.Area.THIS})
@MemoryAreaEncloses(inner = {”this”}, outer = {”handlers”})
@SCJAllowed(javax.safetycritical.annotate.Level.LEVEL_1)
public AperiodicEvent( AperiodicEventHandler []handlers )
```

Constructor for an aperiodic event that is linked to multiple handlers.

Does not allow this or handlers to escape the local variables. Allocates and initializes an array of AperiodicEventHandler within the same scope as this

in order to copy the handlers array. The elements of the handlers array must reside in memory areas that enclose this.

Aside: we do not need to require that "handlers" encloses "this", because we need to make a copy of handlers in order to be robust. However, we do need to required that handlers[i] encloses this for every value of i. Our existing notation does not allow us to say what we might want to say. What I have said is sufficient, but not necessary.

handlers — the handlers that are to be added to this event.

E.2.2 CLASS **AperiodicEventHandler**

DECLARATION

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

CONSTRUCTORS

```
@MemoryAreaEncloses(inner = {"this", "this", "this"}, outer = {"priority", "release_info",
"mem_info"})
@SCJAllowed(javax.safetycritical.annotate.Level.LEVEL_1)
public AperiodicEventHandler( PriorityParameters priority , AperiodicParameters
release_info , StorageParameters scp , long memSize )
```

Constructor to create an aperiodic event handler.

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.

priority — specifies the priority parameters for this periodic event handler. Must not be null.
release_info — specifies the periodic release parameters, in particular the start time, period and deadline miss and cost overrun handlers. 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. TBD whether we support deadline misses and cost overrun detection.
scp — The mem_info parameter describes the organization of memory dedicated to

execution of the underlying thread.
memSize — the size in bytes of the memory area to be used for the execution of this event handler. 0 for an empty memory area. Must not be negative. (added by MS)
Throws IllegalArgumentException if priority, parameters or if memSize is negative.

@MemoryAreaEncloses(inner = {"this", "this", "this", "this"}, outer = {"priority", "release_info", "mem_info", "name"})
@SCJAllowed(javax.safetycritical.annotate.Level.LEVEL_1)
public **AperiodicEventHandler**(PriorityParameters priority , AperiodicParameters release_info , StorageParameters scp , long memSize , String name)

Constructor to create an aperiodic 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_info — specifies the periodic release parameters, in particular the deadline and deadline miss handlers.

scp — The mem_info parameter describes the organization of memory dedicated to execution of the underlying thread.

memSize — the size in bytes of the memory area to be used for the execution of this event handler. 0 for an empty memory area. Must not be negative. (added by MS)
Throws IllegalArgumentException if priority, parameters or if memSize is negative.

E.2.3 CLASS **Cyclet**

TBD: Does the JSR302 expert group approve of the following revision?

A safety-critical application consists of one or more missions, executed concurrently or in sequence. Every Level-0 safety-critical application is represented by Cyclet or a subclass of Cyclet 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.

Given class c of type Cyclet or a subclass of Cyclet that represents a particular SCJ application, the SCJ infrastructure invokes in sequence c.setUp() followed by c.getSequencer(). For the MissionSequencer q returned from s.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. Upon termination of the MissionSequencer's thread, the SCJ infrastructure invokes s.tearDown().

DECLARATION

```
@SCJAllowed
public class Cyclet
  implements javax.safetycritical.Safelet
  extends java.lang.Object
```

CONSTRUCTORS

```
@SCJAllowed
public Cyclet( )
```

Construct a Cyclet.

METHODS

```
@SCJAllowed
@SCJRestricted({javax.safetycritical.annotate.Restrict.INITIALIZATION})
public Level0MissionSequencer  getSequencer( )
```

The default implementation of getSequencer() returns a SingleMissionSequencer() which runs the Level0Mission represented by getPrimordialMission() exactly once. The default sequencer runs at "normal" priority and uses a conservatively large value for StorageParameters.

returns the Level0MissionSequencer that oversees execution of Level0Missions for this application.

```
@SCJAllowed
public void  setUp( )
```

Code to execute before the sequencer starts. The default implementation does nothing.

```
@SCJAllowed
public void tearDown( )
```

Code to execute after the sequencer ends. The default implementation does nothing.

E.2.4 CLASS CyclicExecutive

TBD: An earlier version of CyclicExecutive extended Mission. In the current design, CyclicExecutive produces a MissionSequencer which has the ability to run a sequence of Missions. There's been some back and forth on this. Many of our earlier design choices were based on the assumption that a Level0 Safelet consists of only one Mission, but we subsequently reversed that choice without fixing the relevant libraries. I understand a fundamental desire that "simple things be simple". But there's some question in my mind as to what is simple. The current draft document pursues option 2.

Option 1: CyclicExecutive extends Level0Mission and implements Safelet, with the following consequences:

a. The application developer extends CyclicExecutive b. We need a variant of CyclicExecutive that doesn't extend Level0Mission, because some Level0 applications are going to be sequences of Missions rather than a single mission. c. CyclicExecutive can define a default getSequencer method which returns a SingleMissionSequencer with a "normal" priority and a "reasonably conservative" StorageParameters object, with the single mission represented by "this" CyclicExecutive. d. The user overrides the initialize() and getSchedule() methods, and optionally, the cleanup method. e. I don't like the name CyclicExecutive for this. I'd rather call it CyclicApplication as it is both a Safelet and a Mission.

Option 2: Cyclet is a concrete class that implements Safelet, but does not extend Level0Mission, with the following consequences:

a. Configuraton of the SCJ run-time specifies both the name of the Cyclet subclass (or Cyclet itself) and an optional name of the primordial mission. Infrastructure invokes in sequence the setUp(), getSequencer(), "sequencer.run()", and tearDown(). b. The default implementation of getSequencer returns a SingleMissionSequencer with a normal priority and a reasonably conservative StorageParameters object, representing the single mission that is obtained by invoking the static method of Cyclet that is declared as:

public static Level0Mission getPrimordialMission();

The vendor is required to implement this method in a vendor-specific way. It could, for example, obtain this mission from a command-line argument, or from a configuration choice specified at build time. c. The application developer extends Level0Mission and overrides the initialize() and getSchedule() methods.

DECLARATION

```
@SCJAllowed
public abstract class CyclicExecutive
  implements javax.safetycritical.Safelet
  extends java.lang.Object
```

CONSTRUCTORS

```
@SCJAllowed
@MemoryAreaEncloses(inner = {"this"}, outer = {"storage"})
public CyclicExecutive( StorageParameters storage )
```

Constructor for a Cyclic Executive. Level 0 Applications need to extend CyclicExecutive and define a getSchedule() method. Level 1 and Level 2 applications should not extend CyclicExecutive, but rather should implement Safelet more directly.

storage —

METHODS

```
@SCJAllowed
public static CyclicSchedule  getSchedule( Level0Mission m )
```

TBD: Does the JSR302 expert group approve of the following revision?

A previous revision declared this to be an abstract instance method taking an array of PeriodicEventHandlers as its argument. That earlier design did not generalize to the situation under which a Level0 application consists of a sequence of Missions, each of which needs a distinct cyclic scheduler. This newer design generalizes to sequences of Level-0 missions, and also makes more effective use of the revised design under which Missions reside in the same scope

as their ManagedSchedulables, so a Mission can easily find all of its Managed-Schedulables.

returns the schedule to be used by for the Level0Mission identified by argument m. The cyclic schedule is typically generated by vendor-specific tools. The returned object is expected to reside within the MissionMemory of Level0Mission m.

```
@SCJAllowed
public MissionSequencer getSequencer( )
```

Under normal circumstances, this is invoked from SCJ infrastructure code with ImmortalMemory as the current allocation area.

returns the sequencer to be used for the Level 0 application. By default this is a SingleMissionSequencer, although this method can be overridden by the application if an alternative sequencer is desired.

E.2.5 CLASS **CyclicSchedule**

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

DECLARATION

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

CONSTRUCTORS

```
@Allocate({javax.safetycritical.annotate.Allocate.Area.THIS})
@MemoryAreaEncloses(inner = {"this"}, outer = {"frames"})
@SCJAllowed
public CyclicSchedule( CyclicSchedule.Frame []frames )
```

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

the MissionMemory area that corresponds to the Level0Mission that is to be scheduled.

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 Level0Mission is idle.

E.2.6 CLASS **CyclicSchedule.Frame**

DECLARATION

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

CONSTRUCTORS

```
@Allocate({javax.safetycritical.annotate.Allocate.Area.THIS})
@MemoryAreaEncloses(inner = {"this", "this"}, outer = {"duration", "handlers"})
@SCJAllowed
public CyclicSchedule.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.

E.2.7 CLASS InterruptHandler

DECLARATION

```
@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. It is TBD whether global interrupts are automatically enabled before this method is called.

E.2.8 CLASS InterruptHappening

DECLARATION

```
@SCJAllowed
public class InterruptHappening
  extends javax.realtime.Happening
```

CONSTRUCTORS

```
@SCJAllowed
public InterruptHappening( )
```

```
@SCJAllowed
public InterruptHappening( int id )
```

```
@SCJAllowed
public InterruptHappening( int id , String name )
```

```
@SCJAllowed
public InterruptHappening( String name )
```

METHODS

```
@SCJAllowed
public final int  getPriority( int id )
```

```
@SCJAllowed
protected synchronized void  process( )
```

E.2.9 CLASS Level0Mission

A Level-Zero Safety Critical Java application is comprised of one or more Level0Missions. Each Level0Mission is implemented as a subclass of this abstract Level0Mission class.

DECLARATION

```
@SCJAllowed
public abstract class Level0Mission
  extends javax.safetycritical.Mission
```

CONSTRUCTORS

```
@Allocate({javax.safetycritical.annotate.Allocate.Area.THIS})
@SCJAllowed
public Level0Mission( )
```

Constructor for a Level0Mission. Normally, application-specific code found within the application-defined subclass of MissionSequencer instantiates a new Level0Mission in the MissionMemory area that is dedicated to that Level0Mission. Upon entry into the constructor, this same MissionMemory area is the current allocation area.

Note that this class inherits missionMemorySize(), initialize(), requestTermination(), terminationRequested(), requestSequenceTermination(), sequenceTerminationRequested(), and cleanUp() methods from Mission.

TBD: Under what conditions would we want to prohibit construction of a new Level0Mission. Presumably, it is ”harmless” for a PEH to instantiate a new Level0Mission. But what if the PEH instantiates a Mission, and then tries to ”start” it? Kelvin suggests to resolve this problem by hiding the start method.

E.2.10 CLASS **Level0MissionSequencer**

A MissionSequencer runs a sequence of independent Missions interleaved with repeated execution of certain Missions.

DECLARATION

```
@SCJAllowed
public abstract class Level0MissionSequencer
  extends javax.safetycritical.MissionSequencer
```

CONSTRUCTORS

```
@MemoryAreaEncloses(inner = {”this”}, outer = {”priority”})
@SCJAllowed
```

@SCJRestricted({javax.safetycritical.annotate.Restrict.INITIALIZATION})
public **Level0MissionSequencer**(PriorityParameters priority , StorageParameters storage)

Construct a Level0MissionSequencer to run at the priority and with the memory resources specified by its parameters.

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, or (b) during execution of Mission.initialize() by SCJ infrastructure during initialization of a new Mission in a LevelTwo configuration of the SCJ run-time environment.

METHODS

@SCJAllowed
protected abstract Level0Mission **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() by infrastructure, infrastructure instantiates and enters a very large MissionMemory allocation area. The typical behavior is for getNextMission() to return a Mission object that resides in this MissionMemory area.

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

E.2.11 CLASS **ManagedEventHandler**

DECLARATION

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

METHODS

@SCJAllowed
protected void **cleanUp**( )

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

@SCJAllowed
public String **getName**( )

returns the name of this event handler.

@Override
@SCJAllowed
public abstract void **handleAsyncEvent**( )

Application developers override this method with code to be executed whenever the event(s) to which this event handler is bound is fired.

@Override
@SCJAllowed
public void **register**( )
See Also: javax.safetycritical.ManagedSchedulable.register()

E.2.12 CLASS **ManagedInterruptHappening**

DECLARATION

```
@SCJAllowed
public class ManagedInterruptHappening
  extends javax.safetycritical.InterruptHappening
```

CONSTRUCTORS

@SCJAllowed(javax.safetycritical.annotate.Level.LEVEL_1)
public **ManagedInterruptHappening**( )

Creates a Happening in the current memory area with a system assigned name and id.

@SCJAllowed(javax.safetycritical.annotate.Level.LEVEL_1)
public **ManagedInterruptHappening**(int id)

Creates a Happening in the current memory area with the specified id and a system-assigned name.

@SCJAllowed(javax.safetycritical.annotate.Level.LEVEL_1)
public **ManagedInterruptHappening**(int id , String name)

Creates a Happening in the current memory area with the name and id given.

@SCJAllowed(javax.safetycritical.annotate.Level.LEVEL_1)
public **ManagedInterruptHappening**(String name)

Creates a Happening in the current memory area with the name name and a system-assigned id.

METHODS

@SCJAllowed(javax.safetycritical.annotate.Level.LEVEL_1)
public void **uncaughtException**(Exception E)

Called by the Infrastructure if an interrupt handler throws an uncaught exception

E.2.13 CLASS ManagedMemory

This is the base class for all safety critical Java memory areas. The class provides a uniform method of retrieving the mission manager of the memory areas' mission.

DECLARATION

```
@SCJAllowed
public abstract class ManagedMemory
  extends javax.realtime.LTMemory
```

METHODS

@SCJAllowed
public ManagedSchedulable **getOwner**( )

E.2.14 CLASS **ManagedThread**

DECLARATION

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

CONSTRUCTORS

@MemoryAreaEncloses(inner = {”this”}, outer = {”scheduling”})
@SCJAllowed(javax.safetycritical.annotate.Level.LEVEL_2)
public **ManagedThread**(PriorityParameters scheduling , StorageParameters storage)

> Does not allow this to escape local variables. Creates a link from the constructed object to the scheduling parameter. Thus, scheduling 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.
>
> TBD: what is the ”default” ThreadConfigurationParameters? Or should re remove this constructor?

@MemoryAreaEncloses(inner = {”this”, ”this”, ”this”}, outer = {”schedule”, ”mem_info”, ”logic”})
@SCJAllowed(javax.safetycritical.annotate.Level.LEVEL_2)
public **ManagedThread**(PriorityParameters scheduling , StorageParameters mem_info , Runnable logic)

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

METHODS

```
@SCJAllowed(javax.safetycritical.annotate.Level.LEVEL_2)
public void delay( HighResolutionTime time )
```

```
@SCJAllowed(javax.safetycritical.annotate.Level.LEVEL_2)
public void start( )
```

```
@SCJAllowed(javax.safetycritical.annotate.Level.LEVEL_2)
public boolean terminationPending( )
```

E.2.15 CLASS **Mission**

A Safety Critical Java application is comprised of one or more Missions. Each Mission is implemented as a subclass of this abstract Mission class.

DECLARATION

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

CONSTRUCTORS

```
@Allocate({javax.safetycritical.annotate.Allocate.Area.THIS})
@SCJAllowed(javax.safetycritical.annotate.Level.LEVEL_1)
public Mission( )
```

Constructor for a Mission. Normally, application-specific code found within the application-defined subclass of MissionSequencer instantiates a new Mission in the MissionMemory area that is dedicated to that Mission. Upon entry into the constructor, this same MissionMemory area is the current allocation area.

TBD: Under what conditions would we want to prohibit construction of a new Mission. Presumably, it is ”harmless” for a PEH to instantiate a new Mission. But what if the PEH instantiates a Mission, and then tries to ”start” it? Kelvin suggests to resolve this problem by hiding the start method.

METHODS

@SCJAllowed(javax.safetycritical.annotate.Level.LEVEL_1)
protected void **cleanUp**( )

Method to clean up after an application terminates. Infrastructure calls cleanup after all ManagedSchedulables associated with this Mission have terminated, but before control leaves the dedicated MissionMemory area. The default implementation of cleanUp does nothing. User-defined subclasses may override its implementation.

@SCJAllowed
public static Mission **getCurrentMission**( )

Obtain the current mission.

returns the current mission instance.

@SCJAllowed
protected abstract void **initialize**( )

Perform initialization of the 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. User-defined subclasses may override its implementation.

The typical implementation of initialize() instantiates and registers all ManagedSchedulable objects that consitute 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 Safety Critical Java infrastructure. The infrastructure arranges to begin executing the registered ManagedSchedulable

objects associated with a particular Mission upon return from the 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 MissionMemory scope. Individual ManagedSchedulables can gain access to these objects by passing references to their constructors, or by obtaining a reference to the current mission (by invoking Mission.getCurrentMission()) and coercing this reference to the known Mission subclass.

@SCJAllowed
public abstract long **missionMemorySize**( )

returns the desired size of the MissionMemory assocciated with this Mission. Note that the MissionMemory is allocated initially with a very large size, and then is truncated to the size returned from this method, which is invoked immediately following return from instantiation and construction of this Mission object.

@SCJAllowed
public final void **requestSequenceTermination**( )

Ask for termination of the current mission and its sequencer. The effect of this method is to invoke requestSequenceTermination() on the MissionSequencer that is responsible for execution of this Mission.

TBD: Kelvin made this method final. Ok?

@SCJAllowed
public void **requestTermination**( )

This method provides a standard interface for requesting termination of a Mission. The default implementation has the effect of setting internal state so that subsequent invocations of terminationPending() shall return true. The additional effects are to (1) arrange for all of the periodic event handlers associated with this Mission to be disabled so that no further firings will occur, and (2) arranging to disable all AperiodicEventHandlers so that no further firings will be honored, and (3) decrementing the pending fire count for each event handler so that the event handler can be effectively shut down following completion of any event handling that is currently active.

An application-specific subclass of Mission may override this method in order to insert application-specific code to communicate the intent to shutdown to

specific ManagedSchedulables. It is especially useful to override requestTermination() within Missions that include ManagedThread or inner-nested MissionSequencers.

TBD: there's no mention of pending fire count in the @SCJAllowed API of BoundAsyncEventHandler. What is our intended treatment of this?

```
@SCJAllowed
public final boolean sequenceTerminationPending( )
```

Check if the current MissionSequencer is trying to terminate.

returns true if and only if the requestSequenceTermination() method for the MissionSequencer that controls execution of this Mission has been invoked.

```
@SCJAllowed
public final boolean terminationPending( )
```

Check if the current mission is trying to terminate.

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

E.2.16 CLASS **MissionSequencer**

A MissionSequencer runs a sequence of independent Missions interleaved with repeated execution of certain Missions.

DECLARATION

```
@SCJAllowed
public abstract class MissionSequencer
  extends javax.realtime.BoundAsyncEventHandler
```

CONSTRUCTORS

```
@MemoryAreaEncloses(inner = {"this"}, outer = {"priority"})
@SCJAllowed
@SCJRestricted({javax.safetycritical.annotate.Restrict.INITIALIZATION})
```

public **MissionSequencer**(PriorityParameters priority , StorageParameters storage)

Construct a MissionSequencer to run at the priority and with the memory resources specified by its parameters.

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, or (b) during execution of Mission.initialize() by SCJ infrastructure during initialization of a new Mission in a LevelTwo configuration of the SCJ run-time environment.

METHODS

@SCJAllowed
protected abstract Mission **getNextMission**( )

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

Prior to each invocation of getNextMission() by infrastructure, infrastructure instantiates and enters a very large MissionMemory allocation area. The typical behavior is for getNextMission() to return a Mission object that resides in this MissionMemory 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 synchronized void **handleAsyncEvent**( )

This method is declared final because the implementation is provided by the vendor of the SCJ implementation and shall not be overridden. This method performs all of the activities that correspond to sequencing of Missions by this MissionSequencer.

@SCJAllowed(javax.safetycritical.annotate.Level.LEVEL_2)
public final void **requestSequenceTermination**( )

Try to finish the work of this mission sequencer soon by invoking the currently running Mission's requestTermination method. Upon completion of the currently running Mission, this MissionSequencer shall return from its eventHandler method without invoking getNextMission and without starting any additional missions.

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

TBD: shouldn't we also have a sequenceTerminationPending() method? We need something like this in order to implement Mission.sequenceTerminationPending().

TBD: why restrict this to level_2? in level_0 and level_1, requesting sequence termination represents a mechanism to request "graceful" shutdown of an application.

@SCJAllowed(javax.safetycritical.annotate.Level.LEVEL_2)
public final boolean **sequenceTerminationPending**( )

E.2.17 CLASS NoHeapRealtimeThread

DECLARATION

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

CONSTRUCTORS

@MemoryAreaEncloses(inner = {"this"}, outer = {"scheduling"})
@SCJAllowed(javax.safetycritical.annotate.Level.LEVEL_2)
public **NoHeapRealtimeThread**(PriorityParameters scheduling , StorageParameters mem_info)

Does not allow this to escape local variables. Creates a link from the constructed object to the scheduling parameter. Thus, scheduling 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.

TBD: what is the ”default” ThreadConfigurationParameters? Or should re remove this constructor?

@MemoryAreaEncloses(inner = {”this”, ”this”, ”this”}, outer = {”schedule”, ”mem_info”, ”logic”})
@SCJAllowed(javax.safetycritical.annotate.Level.LEVEL_2)
public **NoHeapRealtimeThread**(PriorityParameters scheduling , StorageParameters mem_info , Runnable logic)

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

METHODS

@SCJAllowed(javax.safetycritical.annotate.Level.LEVEL_2)
public void **start**( )

E.2.18 CLASS **PeriodicEventHandler**

DECLARATION

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

CONSTRUCTORS

@MemoryAreaEncloses(inner = {”this”, ”this”, ”this”}, outer = {”priority”, ”parameters”, ”memSize”})
@SCJAllowed
public **PeriodicEventHandler**(PriorityParameters priority , PeriodicParameters parameters , StorageParameters scp , long memSize)

Constructor to create a periodic event handler.

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.

priority — specifies the priority parameters for this periodic event handler. Must not be null.
parameters — specifies the periodic release parameters, in particular the start time, period and deadline miss and cost overrun handlers. 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.
scp — The scp parameter describes the organization of memory dedicated to execution of the underlying thread. (added by MS)
memSize — the size in bytes of the memory area to be used for the execution of this event handler. 0 for an empty memory area. Must not be negative.
Throws IllegalArgumentException if priority, parameters or if memSize is negative.

@MemoryAreaEncloses(inner = {"this", "this", "this", "this"}, outer = {"priority", "parameters", "memSize", "name"})
@SCJAllowed(javax.safetycritical.annotate.Level.LEVEL_1)
public **PeriodicEventHandler**(PriorityParameters priority , PeriodicParameters release , StorageParameters scp , long memSize , String name)

Constructor to create a periodic 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.

scp — The scp parameter describes the organization of memory dedicated to execution of the underlying thread. (added by MS)

memSize — the size in bytes of the private scoped memory area to be used for the execution of this event handler. 0 for an empty memory area. Must not be negative.

Throws IllegalArgumentException if priority parameters are null or if memSize is negative.

E.2.19 CLASS **PortalExtender**

TBD: what is this?

DECLARATION

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

E.2.20 CLASS **PriorityScheduler**

DECLARATION

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

METHODS

```
@BlockFree
@SCJAllowed
public int  getMaxHardwarePriority( )
```

```
@BlockFree
@SCJAllowed
public int  getMinHardwarePriority( )
```

E.2.21 CLASS **PrivateMemory**

DECLARATION

```
@SCJAllowed
public class PrivateMemory
  extends javax.safetycritical.ManagedMemory
```

CONSTRUCTORS

@SCJAllowed
public **PrivateMemory**(long size)

@SCJAllowed
public **PrivateMemory**(SizeEstimator estimator)

E.2.22 CLASS Services

System wide information

DECLARATION

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

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 dependent manner.

@SCJAllowed
public static AffinitySet **createSchedulingDomain**(BitSet bitSet)

returns an AffinitySet representing the scheduling domain defined by the bitSet
Throws ProcessorAffinityException if a processor indicated by the bitSet already appears in a previously created scheduling domain

@SCJAllowed(javax.safetycritical.annotate.Level.LEVEL_2)
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
TBD: should this be called suspend or deepSleep to no have a ridiculously long name?
TBD: should not be a long nanoseconds?

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

returns the default ceiling priority The value is the highest software priority.

@SCJAllowed
public static Level **getDeploymentLevel**( )

@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 static void **nanoSpin**(int nanos)

Busy wait in nano seconds.

nanos —

@SCJAllowed(javax.safetycritical.annotate.Level.LEVEL_1)
public static void **registerInterruptHandler**(int InterruptId , InterruptHandler IH)

Registers an interrupt handler.

Throws IllegalArgument if unsupported InterruptId IllegalStateException if handler already registered

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

sets the ceiling priority of object O The priority can be in the software or hardware priority range.

Throws IllegalThreadState if called outside the mission phase

E.2.23 CLASS **SingleMissionSequencer**

DECLARATION

```
@SCJAllowed
public class SingleMissionSequencer
  extends javax.safetycritical.Level0MissionSequencer
```

CONSTRUCTORS

@SCJAllowed
@BlockFree
@SCJRestricted({javax.safetycritical.annotate.Restrict.INITIALIZATION})
public **SingleMissionSequencer**(PriorityParameters priority , StorageParameters storage)

METHODS

@SCJAllowed
@BlockFree
@Override
protected Level0Mission **getNextMission**( )
See Also: javax.safetycritical.MissionSequencer.getInitialMission()

E.2.24 CLASS StorageConfigurationParameters

DECLARATION

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

CONSTRUCTORS

@SCJAllowed
public **StorageConfigurationParameters**(long totalBackingStore , int nativeStack , int javaStack)

Stack sizes for schedulable objects and sequencers. Passed as parameter to the constructor of mission sequencers and schedulable objects.

totalBackingStore — size of the backing store reservation for worst-case scope usage in bytes
nativeStack — size of native stack in bytes (vendor specific)
javaStack — size of Java execution stack in bytes (vendor specific)

@SCJAllowed
public **StorageConfigurationParameters**(long totalBackingStore , int nativeStack-Size , int javaStackSize , int messageLength , int stackTraceLength)

Stack sizes for schedulable objects and sequencers. Passed as parameter to the constructor of mission sequencers and schedulable objects.

totalBackingStore — size of the backing store reservation for worst-case scope usage in bytes
nativeStack — size of native stack in bytes (vendor specific)
javaStack — size of Java execution stack in bytes (vendor specific)
messageLength — length of the space in bytes dedicated to message associated with this Schedulable object's ThrowBoundaryError exception plus all the method names/identifiers in the stack backtrace
stackTraceLength — the number of byte for the StackTraceElement array dedicated to stack backtrace associated with this Schedulable object's ThrowBoundaryError exception.

METHODS

@SCJAllowed
public long **getJavaStackSize**( )

returns the size of the Java stack available to the assocated SO.

@SCJAllowed
public int **getMessageLength**( )

return the length of the message buffer

@SCJAllowed
public long **getNativeStackSize**( )

returns the size of the native method stack available to the assocated SO.

@SCJAllowed
public int **getStackTraceLength**( )

return the length of the stack trace buffer

@SCJAllowed
public long **getTotalBackingStoreSize**( )

returns the size of the total backing store available for scoped memory areas created by the assocated SO.

E.2.25 CLASS **StorageParameters**

DECLARATION

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

CONSTRUCTORS

@SCJAllowed
public **StorageParameters**(long totalBackingStore , long nativeStack , long javaStack)

Stack sizes for schedulable objects and sequencers. Passed as parameter to the constructor of mission sequencers and schedulable objects.

TBD: kelvin changed nativeStack and javaStack to long. Note that getJavaStackSize() and getNativeStackSize() methods were already declared to return long. It seems that we have an implicit assumption that memory sizes are represented by long. do others agree with this change?

totalBackingStore — size of the backing store reservation for worst-case scope usage in bytes
nativeStack — size of native stack in bytes (vendor specific)
javaStack — size of Java execution stack in bytes (vendor specific)

@SCJAllowed
public **StorageParameters**(long totalBackingStore , long nativeStackSize , long javaStackSize , int messageLength , int stackTraceLength)

Stack sizes for schedulable objects and sequencers. Passed as parameter to the constructor of mission sequencers and schedulable objects.

TBD: kelvin changed nativeStack and javaStack to long. Note that getJavaStackSize() and getNativeStackSize() methods were already declared to return long. It seems that we have an implicit assumption that memory sizes are represented by long. do others agree with this change?

totalBackingStore — size of the backing store reservation for worst-case scope usage in bytes
nativeStack — size of native stack in bytes (vendor specific)
javaStack — size of Java execution stack in bytes (vendor specific)
messageLength — length of the space in bytes dedicated to message associated with this Schedulable object's ThrowBoundaryError exception plus all the method names/identifiers in the stack backtrace
stackTraceLength — the number of byte for the StackTraceElement array dedicated to stack backtrace associated with this Schedulable object's ThrowBoundaryError exception.

METHODS

@SCJAllowed
public long **getJavaStackSize**( )

returns the size of the Java stack available to the assocated SO.

@SCJAllowed
public int **getMessageLength**( )

return the length of the message buffer

@SCJAllowed
public long **getNativeStackSize**( )

returns the size of the native method stack available to the assocated SO.

@SCJAllowed
public int **getStackTraceLength**( )

return the length of the stack trace buffer

@SCJAllowed
public long **getTotalBackingStoreSize**( )

returns the size of the total backing store available for scoped memory areas created by the assocated SO.

E.2.26 CLASS **Terminal**

A simple Terminal that puts out UTF8 version of String/StringBuilder,.... Does not allocate memory. The output device is implementation dependent and writing to /dev/nul is a a valid implementation.

DECLARATION

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

Author
Martin Schoeberl

METHODS

@SCJAllowed
public static Terminal **getTerminal**( )

Get the single output device.

returns something

```
@SCJAllowed
public void  write( CharSequence s )
```

Write the character sequence to the implementation dependent output device in UTF8.

s —

```
@SCJAllowed
public void  writeln( )
```

Just a CRLF output.

```
@SCJAllowed
public void  writeln( CharSequence s )
```

Same as write, but add a newline. CRLF does not hurt on a Unix terminal.

s —

E.2.27 CLASS **ThrowBoundaryError**

One ThrowBoundaryError is preallocated for each Schedulable in its outermost private scope.

DECLARATION

```
@SCJAllowed
public class ThrowBoundaryError
  extends javax.realtime.ThrowBoundaryError
```

CONSTRUCTORS

@Allocate({javax.safetycritical.annotate.Allocate.Area.CURRENT})
@BlockFree
@SCJAllowed
public **ThrowBoundaryError**( )

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).

METHODS

@BlockFree
@SCJAllowed
public Class **getPropagatedExceptionClass**( )

Performs no allocation. Shall not copy ”this” to any instance or static field.

Returns a reference to the Class of the exception most recently thrown across a scope boundary by the current thread.

@Allocate({javax.safetycritical.annotate.Allocate.Area.CURRENT})
@BlockFree
@SCJAllowed
public String **getPropagatedMessage**( )

Shall not copy ”this” to any instance or static field.

Allocates and returns a 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.

For each \texttt{Schedulable}, a single shared \texttt{String\-Builder} represents the stack back trace method and class names for the most recently constructed \texttt{Throwable}, and the message for the \texttt{Throwable} that most recently crossed a scope boundary. The \texttt{get\-Propagated\-Message} method copies data out of this shared \texttt{StringBuilder} object.

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}.

@Allocate({javax.safetycritical.annotate.Allocate.Area.CURRENT})
@BlockFree
@SCJAllowed
public StackTraceElement [] **getPropagatedStackTrace**( )

Shall not copy ”this” to any instance or static field.

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 ThrowBoundaryError 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. 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.

@BlockFree
@SCJAllowed
public int **getPropagatedStackTraceDepth**( )

Performs no allocation. Shall not copy ”this” to any instance or static field.

Returns the number of valid elements stored within the StackTraceElement array to be returned by getPropagatedStackTrace().

Appendix F

Javadoc Description of Package javax.safetycritical.annotate

Package Contents *Page*

Interfaces

Classes

Level 474

...no description...

Restrict 474

...no description...

F.1 Interfaces

F.2 Classes

F.2.1 CLASS **Level**

DECLARATION

```
@SCJAllowed
public class Level
  extends java.lang.Enum
```

FIELDS

```
@SCJAllowed
public static final Level LEVEL_0
```

```
@SCJAllowed
public static final Level LEVEL_1
```

```
@SCJAllowed
public static final Level LEVEL_2
```

METHODS

```
@SCJAllowed
public static Level  getLevel( String value )
```

```
@SCJAllowed
public abstract int  value( )
```

F.2.2 CLASS **Restrict**

DECLARATION

```
@SCJAllowed
public class Restrict
  extends java.lang.Enum
```

FIELDS

@SCJAllowed
public static final Restrict ALLOCATE_FREE

@SCJAllowed
public static final Restrict ANY_TIME

@SCJAllowed
public static final Restrict BLOCK_FREE

@SCJAllowed
public static final Restrict CLEANUP

@SCJAllowed
public static final Restrict INITIALIZATION

@SCJAllowed
public static final Restrict MAY_ALLOCATE

@SCJAllowed
public static final Restrict MAY_BLOCK

Appendix G

Javadoc Description of Package javax.safetycritical.io

Package Contents *Page*

Interfaces

Classes

Connector 478

The class holding all static methods for creating all connection objects.

ConsoleConnection 479

...no description...

G.1 Interfaces

G.2 Classes

G.2.1 CLASS **Connector**

The class holding all static methods for creating all connection objects.

DECLARATION

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

FIELDS

```
@SCJAllowed
public static final int READ
```

```
@SCJAllowed
public static final int READ_WRITE
```

```
@SCJAllowed
public static final int WRITE
```

METHODS

```
@SCJAllowed
public static Connection  open( String name , int mode )
```

```
@SCJAllowed
public static OutputStream  openOutputStream( String name )
```

G.2.2 CLASS **ConsoleConnection**

DECLARATION

```
@SCJAllowed
public class ConsoleConnection
  implements javax.microedition.io.StreamConnection
  extends java.lang.Object
```

METHODS

```
@SCJAllowed
public void  close( )
```

```
@SCJAllowed
public InputStream  openInputStream( )
```

```
@SCJAllowed
public OutputStream  openOutputStream( )
```

Bibliography