Realtime and Embedded Specification for Java

Version 2.0

Draft 16

Canterbury Edition

17$^{\text{th}}$ of April 2015

Editor
James J. Hunt
aicas GmbH
Haid-und-Neu-Straße 18
D-76131 Karlsruhe, Germany

The Realtime Specification for Java (`RTSJ`) is under development within the Java Community Process (JCP) by the members of the JSR-282 Expert Group (EG). This group, was lead by TimeSys Inc. Corporation, but has been taken over by aicas GmbH.

JSR-282 Expert Group Membership

James J. Hunt	aicas GmbH
Benjamin Brosgol	
Andy Wellings	
Kelvin Nilsen	Atego Systems
Ethan Blanton	

Past Expert Group Members

Peter Dibble	TimeSys
David Holmes	Oracle

Table of Contents

Contents i

List of Figures xii

List of Tables xiii

1 Introduction 1
1.1 Guiding Principles . . . 2
1.1.1 Applicability to Particular Java Environments . . . 2
1.1.2 Backward Compatibility . . . 3
1.1.3 Write Once, Run Anywhere . . . 3
1.1.4 Current Practice vs. Advanced Features . . . 3
1.1.5 Predictable Execution . . . 3
1.1.6 No Syntactic Extension . . . 3
1.1.7 Allow Variation in Implementation Decisions . . . 3
1.1.8 Interoperability . . . 4
1.2 Areas of Enhancement . . . 4
1.2.1 Thread Scheduling and Dispatching . . . 4
1.2.2 Memory Management . . . 5
1.2.3 Synchronization and Resource Sharing . . . 5
1.2.4 Asynchronous Event Handling . . . 5
1.2.5 Task Interruption . . . 5
1.2.6 Raw Memory Access . . . 6
1.2.7 Physical Memory Access . . . 6
1.2.8 Modularization . . . 6

2 Overview 7
2.1 Threads and Scheduling . . . 7
2.2 Synchronization . . . 9
2.2.1 Priority Inversion . . . 9
2.2.2 Priority Inversion Avoidance . . . 10
2.2.3 Execution Eligibility . . . 11

2.2.4 Wait-Free Queues . . . 11
2.3 Asynchrony . . . 11
2.3.1 Asynchronous Events . . . 12
2.3.2 Asynchronous Transfer of Control . . . 13
2.3.3 Principles . . . 13
2.3.3.1 Methodological Principles . . . 14
2.3.3.2 Expressibility Principles . . . 14
2.3.3.3 Semantic Principles . . . 14
2.3.3.4 Pragmatic Principles . . . 15
2.3.4 Asynchronous Realtime Thread Termination . . . 15
2.4 Clocks, Time, and Timers . . . 16
2.5 Memory Management . . . 16
2.5.1 Memory Areas . . . 16
2.5.2 Heap Memory . . . 17
2.5.3 Immortal Memory . . . 17
2.5.4 Scoped Memory . . . 17
2.5.5 Physical Memory Areas . . . 18
2.5.6 Budgeted Allocation . . . 18
2.6 Device Access and Raw Memory . . . 19
2.6.1 Raw Memory Access . . . 19
2.7 System Options . . . 19
2.8 Exceptions . . . 20
2.9 Summary . . . 20

3 Requirements and Conventions **21**
3.1 Base Requirements . . . 21
3.2 Modules . . . 22
3.2.1 Base Module . . . 22
3.2.2 Device Module . . . 24
3.2.3 Alternative Memory Areas Module . . . 25
3.2.4 Optional Features . . . 26
3.2.5 Deprecated Classes . . . 27
3.3 Conditionally-Required Facilities . . . 28
3.3.1 Options for Development Platforms . . . 28
3.4 Required Documentation . . . 29
3.5 Conventions . . . 30
3.6 Definitions . . . 30

4 Conventional Java Classes and Language **33**
4.1 Scheduling . . . 33
4.1.1 Priority . . . 34

4.1.1.1 Setting Priority 34
4.1.1.2 Getting Priority 35
4.1.2 Thread Groups 35
4.1.3 Current Thread 36
4.2 Java Memory Model 36
4.3 InterruptedException 37
4.4 Memory Management 37
4.4.1 Memory Areas 37
4.4.2 Garbage Collection 38
4.4.3 Realtime Garbage Collections 38
4.4.3.1 Thread-Based Collectors 38
4.4.3.2 Allocation-Based Collectors 38
4.4.3.3 Alternatives to Garbage Collection 39
4.4.3.4 Developer Implementation 39
4.5 Summary 39

5 Realtime Threads **41**
5.1 Overview 41
5.2 Semantics 42
5.3 Package javax.realtime 44
5.3.1 Enumerations 44
5.3.1.1 PhasingPolicy 44
5.3.2 Classes 46
5.3.2.1 ConfigurationParameters 46
5.3.2.2 RealtimeThread 51
5.4 Rationale 77

6 Scheduling **79**
6.1 Overview 79
6.2 Definitions 80
6.3 Semantics 82
6.3.1 Schedulers 82
6.3.2 The Base Scheduler 83
6.3.2.1 Priorities 83
6.3.2.2 Dispatching 84
6.3.2.3 Parameter Values 86
6.3.2.4 Cost Monitoring and Cost Enforcement 87
6.3.2.5 Release of Realtime Threads 89
6.3.2.6 Aperiodic Release Control 100
6.3.2.7 Sporadic Release Control 101
6.3.2.8 Release Control for Asynchronous Event Handlers 103

6.3.2.9 Processing Groups 104
6.4 Package javax.realtime 107
6.4.1 Interfaces 107
6.4.1.1 BoundSchedulable 107
6.4.1.2 Schedulable 107
6.4.2 Classes 116
6.4.2.1 Affinity 116
6.4.2.2 AperiodicParameters 128
6.4.2.3 ImportanceParameters 135
6.4.2.4 PeriodicParameters 138
6.4.2.5 PriorityParameters 144
6.4.2.6 PriorityScheduler 146
6.4.2.7 ProcessingGroupParameters 151
6.4.2.8 ReleaseParameters 159
6.4.2.9 Scheduler 166
6.4.2.10 SchedulingParameters 169
6.4.2.11 SporadicParameters 171
6.5 Rationale 177
6.5.1 Multiprocessor Support 178
6.5.2 Impact of Clock Granularity 179
6.5.3 Deadline Miss Detection 180

7 Synchronization 181
7.1 Overview 181
7.2 Semantics 181
7.2.1 The Base Priority Scheduler 182
7.2.2 Additional Schedulers 184
7.3 Package javax.realtime 186
7.3.1 Classes 186
7.3.1.1 MonitorControl 186
7.3.1.2 PriorityCeilingEmulation 189
7.3.1.3 PriorityInheritance 192
7.3.1.4 WaitFreeDequeue 193
7.3.1.5 WaitFreeReadQueue 193
7.3.1.6 WaitFreeWriteQueue 201
7.4 Rationale 207

8 Asynchrony 209
8.1 Overview 209
8.2 Definitions 211
8.3 Semantics 212

8.3.1 Asynchronous Events and their Handlers . . . 212
8.3.2 Active Events and Dispatching . . . 214
8.3.3 Asynchronous Transfer of Control . . . 215
8.3.3.1 Summary of ATC Operation . . . 218
8.4 Package javax.realtime . . . 220
8.4.1 Interfaces . . . 220
8.4.1.1 ActiveEvent . . . 220
8.4.1.2 BoundAbstractAsyncEventHandler . . . 222
8.4.1.3 Interruptible . . . 222
8.4.2 Exceptions . . . 224
8.4.2.1 AsynchronouslyInterruptedException . . . 224
8.4.2.2 Timed . . . 229
8.4.3 Classes . . . 232
8.4.3.1 AbstractAsyncEvent . . . 232
8.4.3.2 AbstractAsyncEventHandler . . . 236
8.4.3.3 ActiveEventDispatcher . . . 247
8.4.3.4 AsyncEvent . . . 249
8.4.3.5 AsyncEventHandler . . . 251
8.4.3.6 AsyncLongEvent . . . 257
8.4.3.7 AsyncLongEventHandler . . . 259
8.4.3.8 AsyncObjectEvent . . . 263
8.4.3.9 AsyncObjectEventHandler . . . 265
8.4.3.10 BoundAsyncEventHandler . . . 270
8.4.3.11 BoundAsyncLongEventHandler . . . 272
8.4.3.12 BoundAsyncObjectEventHandler . . . 274
8.5 Rationale . . . 276

9 Time 279
9.1 Overview . . . 279
9.2 Definitions . . . 279
9.3 Semantics . . . 280
9.4 Package javax.realtime . . . 283
9.4.1 Classes . . . 283
9.4.1.1 AbsoluteTime . . . 283
9.4.1.2 HighResolutionTime . . . 297
9.4.1.3 RelativeTime . . . 306
9.5 Rationale . . . 316

10 Clocks and Timers 317
10.1 Overview . . . 317
10.2 Definitions . . . 318

10.3 Semantics . . . 319
10.3.1 Clock Model . . . 319
10.3.2 Clocks and Timables . . . 320
10.3.3 Timers . . . 323
10.3.3.1 Counter Model . . . 324
10.3.3.2 Comparator Model . . . 324
10.3.3.3 Triggering . . . 324
10.3.3.4 Behavior of Timers . . . 325
10.3.3.5 Phasing . . . 326
10.4 Package javax.realtime . . . 327
10.4.1 Interfaces . . . 327
10.4.1.1 AsyncTimable . . . 327
10.4.1.2 Chronograph . . . 327
10.4.1.3 Timable . . . 330
10.4.2 Classes . . . 331
10.4.2.1 Clock . . . 331
10.4.2.2 OneShotTimer . . . 337
10.4.2.3 PeriodicTimer . . . 339
10.4.2.4 TimeDispatcher . . . 348
10.4.2.5 Timer . . . 350
10.5 Rationale . . . 371

11 Alternative Memory Areas 373
11.1 Overview . . . 373
11.1.1 Physical Memory . . . 374
11.1.2 Stacked Memory . . . 375
11.1.3 Summary . . . 378
11.2 Definitions . . . 379
11.3 Semantics . . . 380
11.3.1 Allocation time . . . 380
11.3.2 The allocation context . . . 380
11.3.3 The Parent Scope . . . 381
11.3.4 Memory areas and schedulables . . . 382
11.3.5 Scoped memory reference counting . . . 382
11.3.6 Immortal memory . . . 384
11.3.7 Maintaining referential integrity . . . 384
11.3.8 Object initialization . . . 385
11.3.9 Maintaining the Scope Stack . . . 385
11.3.10 The `enter` method . . . 386
11.3.11 The `executeInArea` or `newInstance` methods . . . 386
11.3.12 Constructor methods for Schedulables . . . 387

11.3.13 The Single Parent Rule . . . 387
11.3.14 Scope Tree Maintenance . . . 388
11.3.14.1 On Scope Stack Push of ma . . . 388
11.3.14.2 On Scope Stack Pop of ma . . . 388
11.4 Package javax.realtime . . . 390
11.4.1 Interfaces . . . 390
11.4.1.1 ChildScopeVisitor . . . 390
11.4.1.2 PhysicalMemoryCharacteristic . . . 391
11.4.1.3 PhysicalMemoryFilter . . . 391
11.4.1.4 VirtualMemoryCharacteristic . . . 392
11.4.2 Enumerations . . . 392
11.4.2.1 NewPhysicalMemoryManager.CachingBehavior . . . 392
11.4.2.2 NewPhysicalMemoryManager.PagingBehavior . . . 393
11.4.3 Classes . . . 394
11.4.3.1 HeapMemory . . . 394
11.4.3.2 ImmortalMemory . . . 399
11.4.3.3 ImmortalPhysicalMemory . . . 401
11.4.3.4 LTMemory . . . 405
11.4.3.5 LTPhysicalMemory . . . 412
11.4.3.6 MemoryArea . . . 416
11.4.3.7 MemoryParameters . . . 429
11.4.3.8 NewPhysicalMemoryManager . . . 433
11.4.3.9 PhysicalMemoryModule . . . 437
11.4.3.10 PinnableMemory . . . 439
11.4.3.11 ScopedMemory . . . 442
11.4.3.12 SizeEstimator . . . 466
11.4.3.13 StackedMemory . . . 470
11.5 The Rationale . . . 481
11.5.1 The scoped memory model . . . 481
11.5.2 The physical memory model . . . 482
11.5.2.1 Problems with the current RTSJ 1.0.2 Physical Memory Framework . . . 485
11.5.2.2 The RTSJ 2.0 Physical Memory Framework . . . 486
11.5.2.3 An example . . . 487

12 Devices and Triggering 491
12.1 Overview . . . 491
12.2 Definitions . . . 492
12.3 Semantics . . . 493
12.3.1 Raw Memory . . . 493
12.3.1.1 Raw Memory Region . . . 495

12.3.1.2 Raw Memory Factory . . . 496
12.3.1.3 Stride . . . 496
12.3.2 Direct Memory Access Support . . . 496
12.3.3 External Triggering . . . 497
12.3.3.1 Happenings . . . 497
12.3.4 Interrupt Service Routines . . . 499
12.4 Package javax.realtime.device . . . 503
12.4.1 Interfaces . . . 503
12.4.1.1 RawByte . . . 503
12.4.1.2 RawByteReader . . . 503
12.4.1.3 RawByteWriter . . . 506
12.4.1.4 RawDouble . . . 508
12.4.1.5 RawDoubleReader . . . 509
12.4.1.6 RawDoubleWriter . . . 512
12.4.1.7 RawFloat . . . 514
12.4.1.8 RawFloatReader . . . 515
12.4.1.9 RawFloatWriter . . . 517
12.4.1.10 RawInt . . . 520
12.4.1.11 RawIntReader . . . 520
12.4.1.12 RawIntWriter . . . 523
12.4.1.13 RawLong . . . 526
12.4.1.14 RawLongReader . . . 526
12.4.1.15 RawLongWriter . . . 529
12.4.1.16 RawMemory . . . 532
12.4.1.17 RawMemoryRegionFactory . . . 532
12.4.1.18 RawShort . . . 552
12.4.1.19 RawShortReader . . . 552
12.4.1.20 RawShortWriter . . . 555
12.4.2 Exceptions . . . 557
12.4.2.1 UnsupportedRawMemoryRegionException . . . 557
12.4.3 Classes . . . 559
12.4.3.1 Happening . . . 559
12.4.3.2 HappeningDispatcher . . . 565
12.4.3.3 InterruptServiceRoutine . . . 567
12.4.3.4 RawBufferFactory . . . 571
12.4.3.5 RawMemoryFactory . . . 575
12.4.3.6 RawMemoryRegion . . . 598
12.5 Rationale . . . 600
12.5.1 Raw Memory . . . 600
12.5.1.1 Direct memory access . . . 602

12.5.2 Interrupt Handling . . . 603
12.5.3 An Illustrative Example . . . 605
12.5.3.1 Software architecture . . . 605
12.5.3.2 Device initialization . . . 607
12.5.3.3 Responding to external happenings . . . 608
12.5.3.4 Access to the flash controller's device registers . . . 609

13 System and Options **611**
13.1 Overview . . . 611
13.2 Semantics . . . 611
13.2.0.5 RealtimeSystem . . . 612
13.2.0.6 RealtimeSecurity . . . 612
13.2.0.7 POSIX Signals . . . 612
13.2.0.8 POSIX Realtime Signals . . . 612
13.3 Package javax.realtime . . . 613
13.3.1 Classes . . . 613
13.3.1.1 GarbageCollector . . . 613
13.3.1.2 POSIXRealtimeSignal . . . 614
13.3.1.3 POSIXRealtimeSignalDispatcher . . . 619
13.3.1.4 POSIXSignal . . . 621
13.3.1.5 POSIXSignalDispatcher . . . 626
13.3.1.6 RealtimeSecurity . . . 628
13.3.1.7 RealtimeSystem . . . 632
13.4 Rationale . . . 637

14 Exceptions **639**
14.1 Overview . . . 639
14.1.1 Semantics . . . 639
14.2 Package javax.realtime . . . 640
14.2.1 Interfaces . . . 640
14.2.1.1 PreallocatedThrowable . . . 640
14.2.2 Exceptions . . . 644
14.2.2.1 ArrivalTimeQueueOverflowException . . . 644
14.2.2.2 CeilingViolationException . . . 645
14.2.2.3 DeregistrationException . . . 647
14.2.2.4 DuplicateEventException . . . 648
14.2.2.5 DuplicateFilterException . . . 649
14.2.2.6 InaccessibleAreaException . . . 650
14.2.2.7 LateStartException . . . 651
14.2.2.8 MITViolationException . . . 652
14.2.2.9 MemoryInUseException . . . 653

14.2.2.10 MemoryScopeException . . . 654
14.2.2.11 MemoryTypeConflictException . . . 655
14.2.2.12 OffsetOutOfBoundsException . . . 656
14.2.2.13 ProcessorAffinityException . . . 658
14.2.2.14 RegistrationException . . . 658
14.2.2.15 ScopedCycleException . . . 660
14.2.2.16 SizeOutOfBoundsException . . . 661
14.2.2.17 UnknownHappeningException . . . 663
14.2.2.18 UnsupportedPhysicalMemoryException . . . 664
14.2.3 Classes . . . 665
14.2.3.1 AlignmentError . . . 665
14.2.3.2 BacktraceManagement . . . 666
14.2.3.3 IllegalAssignmentError . . . 668
14.2.3.4 MemoryAccessError . . . 670
14.2.3.5 ResourceLimitError . . . 671
14.2.3.6 ThrowBoundaryError . . . 672
14.2.4 Rationale . . . 673

15 Deprecated Classes 675
15.1 Overview . . . 675
15.2 Semantics . . . 675
15.3 Package javax.realtime . . . 676
15.3.1 Interfaces . . . 676
15.3.1.1 PhysicalMemoryName . . . 676
15.3.1.2 PhysicalMemoryTypeFilter . . . 676
15.3.2 Classes . . . 683
15.3.2.1 NoHeapRealtimeThread . . . 683
15.3.2.2 POSIXSignalHandler . . . 686
15.3.2.3 PhysicalMemoryManager . . . 694
15.3.2.4 RationalTime . . . 704
15.3.2.5 RawMemoryAccess . . . 704
15.3.2.6 RawMemoryFloatAccess . . . 729
15.3.2.7 VTMemory . . . 740
15.3.2.8 VTPhysicalMemory . . . 746
15.4 Rationale . . . 762

16 Conformance, Compliance, and Portability 763
16.1 Minimum Implementations . . . 763
16.2 Modules . . . 763
16.3 Optionally Required Components . . . 763
16.3.1 Deployment Implementation . . . 764

16.4 Simulation Implementation 765
16.5 Documentation Requirements 765

A Bibliography **767**

List of Figures

8.1 The Event Class Hierarchy . . . 212
8.2 States of a Simple AbstractAsyncEvent . . . 214
8.3 States of an ActiveEvent . . . 216

10.1 Sequence Diagram for Using a Timer . . . 322
10.2 Sequence Diagram for Realtime Sleep . . . 323
10.3 States of a Timer . . . 326

11.1 Manipulation of `StackedMemory` Areas . . . 377

12.1 Raw Memory Interface . . . 494
12.2 Event Classes . . . 495
12.3 Happening State Transition Diagram . . . 498
12.4 Interrupt servicing . . . 499
12.5 Creating Raw Memory Accessors . . . 601
12.6 Flash memory device . . . 603
12.7 Flash memory classes . . . 607
12.8 Sequence diagram showing initialization operations . . . 607
12.9 Sequence diagrams showing operations to initialize the hardware device 608
12.10 The `FMSocketController.handleAsync` method . . . 609
12.11 Application usage . . . 610

List of Tables

3.1 RTSJ Options 26

5.1 Effect of PhasingPolicy on the First Release of a RealtimeThread with PeriodicParameters 45

12.1 Device registers 606

Chapter 1

Introduction

The goal of the *Real-Time Specification for Java* (`RTSJ`) is to support the use of Java technology in embedded and realtime systems. It provides a specification for refining the *Java Language Specification* and the *Java Virtual Machine Specification* and of providing an extended Application Programming Interface that facilitates the creation, verification, analysis, execution, and management of realtime Java programs such as control and sensor applications.

The Java Virtual Machine and the Java Language were conceived as a portable environment for desktop and server applications. The emphasis has been on throughput and responsiveness. These are characteristics obtainable with time-sharing systems. For this conventional Java environment, it is more important that each task makes progress, than that a particular task completes within a predefined time slot.

In a realtime system, the system tries to schedule the most critical task that is ready to run first. This task runs either until it is finished, or it needs to wait for some event or data, or a more critical task is released or a more critical task becomes schedulable after waiting for its event or data.

Realtime scheduling is commonly done with a priority preemptive scheduler, where tasks that have short deadlines are given higher priority than tasks that have longer deadlines. The programmer is responsible for encoding some notion of task importance to priorities. The goal is to see that all tasks finish within their deadlines. Scheduling analysis, such as Rate Monotonic Analysis, can be used to help determine this.

Many realtime systems have nonrealtime components, so it is desirable to be able to combine realtime and nonrealtime tasks in a single system. Realtime tasks are then given preference over nonrealtime tasks. For Java, this means that realtime tasks must be scheduled before threads with conventional Java priorities (1–10). Being able to synchronize between tasks, both realtime and conventional Java threads, adds additional requirements.

Providing realtime semantics and the additional programming interfaces required is a core part of this specification. So much so that the original specification provided special memory areas to avoid the use of garbage collection. The availability of various techniques for realtime garbage collection has changed the state of practice since RTSJ Version 1.0. Though still part of the specification, these special memory areas are no longer central to it. Realtime scheduling and priority inversion avoidance for synchronization are the core of providing realtime response. These are provided through refinements to the base Java semantics and additional classes.

Realtime tasks can be modeled both with realtime threads and with event handlers. Realtime threads are much the same as conventional Java threads except for how they are scheduled. Event handlers encapsulate a bit of work that is done every time some event occurs. Events are referred to as asynchronous because they generally occur independent of program flow. Thus, a timed event is considered to be an asynchronous event, but scheduled periodically. Event handling provides a less resource intensive means of writing control applications because the underlying thread mechanism can be shared between event handlers. Deadline analysis is also somewhat simpler because the end of the work to be done is well bounded. Event handling is ideal for periodic tasks and responding to external impulses. The specification provides both paradigms.

Though realtime is necessary for many control tasks, it is not sufficient. A significant part of the RTSJ API addresses communication with the outside world through devices and signals. This makes it possible to write control applications without resorting to JNI, thereby maintaining the integrity and safety that Java offers.

Since not all applications need all aspects of the specification, there are now modules to suite the major application scenarios. This should make it easier for conventional JVM providers to include basic specification facilities without negatively impacting their core application domains, but still be compatible with hard realtime implementations. The goal is to make the transition between conventional JVMs and realtime JVMs easier.

1.1 Guiding Principles

Providing a coherent semantics and set of programming interfaces requires some guiding principles around which to organize the RTSJ. These principles delimit the scope of the RTSJ and its compatibility requirements with conventional Java.

1.1.1 Applicability to Particular Java Environments

The RTSJ shall not include specifications that restrict its use to a particular Java environment, such as a particular versions of the Java Development Kit, an Embedded

Java Application Environment, or a Java Edition, beyond the natural development of the Java language.

1.1.2 Backward Compatibility

The `RTSJ` shall not prevent existing, properly written, conventional Java programs from executing on implementations of the `RTSJ`.

1.1.3 Write Once, Run Anywhere

The `RTSJ` should recognize the importance of "Write Once, Run Anywhere", but it should also recognize the difficulty of achieving WORA for realtime programs and not attempt to increase or maintain binary portability at the expense of predictability. Hence, the goal should be "Write Once Carefully, Run Anywhere Conditionally".

1.1.4 Current Practice vs. Advanced Features

The `RTSJ` should address current realtime system practice as well as allow future implementations to include advanced features.

1.1.5 Predictable Execution

The `RTSJ` shall hold predictable execution as first priority in all trade-offs; this may sometimes be at the expense of typical general-purpose computing performance measures.

1.1.6 No Syntactic Extension

In order to facilitate the job of tool developers, and thus to increase the likelihood of timely implementations, the `RTSJ` shall not introduce new keywords or make other syntactic extensions to the Java language.

1.1.7 Allow Variation in Implementation Decisions

Implementations of the `RTSJ` may vary in a number of implementation decisions, such as the use of efficient or inefficient algorithms, trade-offs between time and space efficiency, inclusion of scheduling algorithms not required in the minimum implementation, and variation in code path length for the execution of byte codes. The `RTSJ` should not mandate algorithms or specific time constants for such, but require that the semantics of the implementation be met and where necessary put limits on execution time complexity. The `RTSJ` offers implementers the flexibility to create implementations suited to meet the requirements of their customers.

1.1.8 Interoperability

It should be possible to implement all aspects of the `RTSJ` on a conventional JVM with the exception that realtime response and pointer assignment rules would not necessarily be guaranteed. This should ease the transition between conventional and realtime programming and aid functional testing on a conventional JVM. The API should support modules for this as well.

1.2 Areas of Enhancement

Each guiding principle has had a direct effect on the development of the specification. There are eight aspects of these refinements and additions in the specification. Their enumeration should aid the understanding of the rest of the specification.

1.2.1 Thread Scheduling and Dispatching

Portability dictates the specification of at least one standard realtime scheduler, but in light of the significant diversity in scheduling and dispatching models and the recognition that each model has wide applicability in the diverse realtime systems industry, the specification provides an underlying scheduling infrastructure that can be extended to use other scheduling algorithms for scheduling realtime Java threads and event handlers.

To accommodate current practice, the `RTSJ` shall require a base scheduler in all implementations. The required base scheduler will be familiar to realtime system programmers. It is a priority preemptive scheduler with priorities above the conventional Java priorities (1–10).

The specification is constructed to allow implementations to provide unanticipated scheduling algorithms. Implementations will enable the programmatic assignment of parameters appropriate for the underlying scheduling mechanism as well as provide any necessary methods for the creation, management and termination of realtime Java threads. In the current specification, any other thread, scheduling, and dispatching mechanism may be bound to an implementation; however, there should be enough flexibility in the thread scheduling framework to enable future versions of the specification to build on this release.

1.2.2 Memory Management

Automatic memory management is a particularly important feature of the Java programming environment. The specification enables, as far as possible, the job of memory management to be implemented automatically by the underlying system and not intrude on the programming task. Many automatic memory management algorithms, also known as garbage collection (GC), exist, and many of those apply to certain classes of realtime programming styles and systems. In an attempt to accommodate a diverse set of GC algorithms, the specification defines a memory allocation and reclamation paradigm that

- is independent of any particular GC algorithm,
- requires the VM to precisely characterize its GC algorithm's effect on the preemption of realtime Java tasks, and
- enables the allocation and reclamation of objects outside of any interference by any GC algorithm.

1.2.3 Synchronization and Resource Sharing

Logic often requires serial access to resources, and realtime systems introduce an additional complexity: the need to minimize priority inversion and hence the excessive delay of more critical tasks. The least intrusive specification for enabling realtime safe synchronization is to require that implementations of the Java keyword `synchronized` implement one or more algorithms that prevent priority inversion among realtime Java tasks that share the serialized resource. In addition, the specification provides other data passing mechanisms to minimize the need for synchronization.

1.2.4 Asynchronous Event Handling

Realtime systems typically interact closely with the real world. With respect to the execution of logic, the real world is asynchronous; therefore, the specification includes efficient mechanisms for programming disciplines that would accommodate this inherent asynchrony. The `RTSJ` has a general mechanism for asynchronous event handling. This specification provides classes that represent things that can happen and logic that executes when those things happen. The execution of the logic is scheduled and dispatched by the `RTSJ` runtime.

1.2.5 Task Interruption

Sometimes, the real world changes so drastically (and asynchronously) that the current point of logic execution should be immediately, efficiently, and safely ended and control transferred to another point of execution. The `RTSJ` provides a mechanism which extends Java's interrupt and exception handling mechanisms to enable applications to programmatically change the locus of control of another Java task. This mechanism may restrict this asynchronous transfer of control to logic specifically written with the assumption that its locus of control may asynchronously change. Due to the inherent susceptibility to deadlock, the `Thread.stop` method cannot be used for this.

1.2.6 Raw Memory Access

Accessing device memory is not in and of itself a realtime issue, many realtime systems require it for providing realtime control of a system. This requires an API providing programmers with byte-level access to physical device registers, whether in main memory or in some I/O space. This API must be as efficient as possible, since such access is often under tight time constraints.

1.2.7 Physical Memory Access

Some systems provide memory areas that differ in important aspects, such as time to read or write data and its persistence. Being able to take advantage of these areas can have an impact on performance. This specification enables their efficient use.

1.2.8 Modularization

Not all applications require all aspects of the specification. In fact, having a core set of the APIs presented is useful for conventional Java programming and aids overall interoperability. To this end, the specification provides a core set of APIs and a few optional modules as well as semantics for use in conventional JVMs that do not offer realtime guarantees. This should enable implementations to be optimized for particular use cases and enable conventional Java environments to be used to help develop code that can be more easily shared between realtime and conventional systems.

Chapter 2

Overview

The RTSJ comprises several areas of extended semantics. These areas are discussed in approximate order of their relevance to realtime programming. The semantics and mechanisms of each of threads and scheduling, synchronization, asynchrony, clocks and timers, memory management, device access and raw memory, system options, and exceptions are all crucial to the acceptance of the RTSJ as a viable realtime development platform. Further details, exact requirements, class documentation, and rationale for these extensions are given in subsequent chapters.

2.1 Threads and Scheduling

One of the concerns of realtime programming is to ensure the timely and predictable execution of sequences of machine instructions. Various scheduling schemes name these sequences of instructions differently, for example, thread, task, module, or block. In Java, this computation is executed in the context of a thread. Since Java threads were designed for fair execution[1] rather than predictable execution, the RTSJ introduces the concept of a *schedulable*. These are the objects managed by the base scheduler: RealtimeThread and its subclasses and AbstractAsyncEventHandler and its subclasses. `RealtimeThread` is a specialization of Java's `Thread`.

Timely execution of schedulables means that the programmer can determine, by analysis of the program, testing the program on particular implementations, or both, whether particular threads will always complete execution before a given timeliness constraint. This is the essence of realtime programming: the addition of temporal constraints to the correctness conditions for computation. For example, for a program to compute the sum of two numbers, it may no longer be acceptable to

[1] Actually, neither the Java Virtual Machine Specification[6] nor the Java Language Specification[5] defines how Java threads should be scheduled, but most implementations, including the reference implementations, use some sort of fair scheduling.

compute only the correct arithmetic answer but the answer must be computed within a particular time interval. Typically, temporal constraints are deadlines expressed in either relative or absolute time.

The term *scheduling* (or *scheduling algorithm*) refers to the production of a sequence (or ordering) for the execution of a set of schedulables (a *schedule*). This schedule attempts to optimize a particular metric (a metric that measures how well the system is meeting the temporal constraints). A *feasibility analysis* determines if a schedule has an acceptable value for the metric. For example in hard realtime systems, the typical metric is "number of missed deadlines" and the only acceptable value for that metric is zero. So called soft realtime systems use other metrics (such as mean tardiness) and may accept various values for the metric in use.

Many systems, including most conventional Java implementations, use thread priority to guide the determination of a schedule. Priority is typically an integer associated with a thread; these integers convey to the system the order in which the threads should execute. The generalization of the concept of priority is *execution eligibility*. The term *dispatching* refers to that portion of the system which selects the thread with the highest execution eligibility from the pool of threads that are ready to run.

In current realtime system practice, the assignment of priorities is typically under programmer control as opposed to under system control. As a base scheduler for realtime tasks, the `RTSJ` provides preemptive priority-based first-in-first-out (FIFO) scheduler, which also leaves the assignment of priorities to programmer control. Most realtime operation systems (RTOS) are also based on priority preemptive FIFO scheduling.

The `RTSJ` defines a number of classes with names of the format `<string>Parameters` such as `ReleaseParameters`, which provide parameters for resource management. An instance of one of these parameter classes holds a particular resource-demand characteristic for one or more schedulables. For example, the `PriorityParameters` subclass of `SchedulingParameters` contains the execution eligibility metric of the base scheduler, i.e., a priority. At some time (construction-time or later when the parameters are replaced using setter methods), instances of parameter classes are bound to a schedulable. The schedulable then assumes the characteristics of the values in the parameter object. For example, a `PriorityParameters` instance with its priority set to the value representing the highest priority available on a system is bound to a schedulable, then that schedulable will assume the characteristic that it will execute whenever it is ready in preference to all other schedulables (except, of course, those also with the same priority).

The `RTSJ` provides implementers with the flexibility to install arbitrary scheduling algorithms in an implementation of the specification. This is to support the widely varying requirements of the realtime systems industry with respect to scheduling. Use of the Java platform may help produce code written once but able to be executed on many different computing platforms. The `RTSJ` contributes to this goal, but the rigors of realtime systems detract from it. The `RTSJ`'s rigorous specification of the required priority scheduler is critical for portability of time-critical code, but the `RTSJ` permits and supports platform-specific schedulers which are not necessarily portable.

2.2 Synchronization

If the computation in each thread were independent of the computation in all other threads, scheduling alone would be enough to ensure timeliness; however, this is usually not the case. Threads often need to communicate with one another or share data. Resources must be shared as well. Two threads cannot read different data from the disk at the same time nor write data to a disk at the same time. They cannot send a message to another machine at the same time. They cannot update the same in-memory data at the same time. One thread may have to wait for another thread to get the data it needs. Just as in a normal system, synchronization is required. In a realtime system, this synchronization must not prevent other threads from completing their tasks on time.

2.2.1 Priority Inversion

The additional concern for synchronization in a realtime system, as opposed to a conventional system, is that blocking can cause the wrong thread to run first. A high priority thread can be blocked by a low priority thread that is vying for the same resource. A priority queue can be used to ensure that a highest priority thread goes first, when more than one thread is waiting to enter a synchronized block, but this is not always sufficient.

Consider a single processor system with three threads, t_1, t_2, and t_3, where t_1 has the highest priority and t_3 has the lowest priority. It is possible that t_2 can prevent t_1 from running by preempting t_3. This is called priority inversion. It occurs when t_1 is blocked by attempting to acquire a lock that is held by thread t_3 and t_3 is preempted by t_2. When t_2 does run, it may prevent t_3 from running indefinitely, thereby keeping t_1 blocked past its deadline.

What is needed is a mechanism the ensure that, while t_1 is waiting on a resource in use by t_3, thread t_3 runs before all threads with a priority less than that of t_1.

2.2.2 Priority Inversion Avoidance

Two of the most common mechanisms for avoiding priority inversion are priority inheritance and priority ceiling emulation (a.k.a. highest locker protocol). Both of these boost the priority of a thread holding the lock in order to prevent a noncontending thread from transitively blocking a higher priority thread which is waiting for the same lock. The difference is how high the priority is raised and when. Both take effect when a thread is in a `synchronized` section of code.

The first is the default behavior for `synchronized` blocks and methods. It applies to all code running within the implementation, not just to `schedulables`. The priority inheritance protocol is a well-known algorithm in the realtime scheduling literature and it has the following effect. If thread t_1 attempts to acquire a lock that is held by a lower-priority thread t_3, then t_3's priority is raised to that of t_1 as long as t_3 holds the lock (and recursively if t_3 is itself waiting to acquire a lock held by an even lower-priority thread).

The specification also provides a mechanism by which the programmer can override the default system-wide policy, or control the policy to be used for a particular monitor, provided that policy is supported by the implementation. The second policy, priority ceiling emulation protocol, can be set using this mechanism. It is also a well-known algorithm in the literature. The following three points provide a somewhat simplified description of its effect.

- A monitor is given a "priority ceiling" when it is created; the programmer should choose at least the highest priority of any thread that could attempt to enter the monitor.
- As soon as a thread enters synchronized code, its (active) priority is raised to the monitor's ceiling priority. If, through programming error, a thread has a higher base priority than the ceiling of the monitor it is attempting to enter, then an exception is thrown.
- On leaving the monitor, the thread has its active priority reset. In simple cases it will set be to the thread's previous active priority, but under some circumstances (e.g. a dynamic change to the thread's base priority while it was in the monitor) a different value is possible.

In addition, threads and asynchronous event handlers waiting to acquire a resource must be released from highest to lowest priority (in priority order). This applies to processors as well as to synchronized blocks. If schedulables with the same priority are possible under the active scheduling policy, such schedulables are awakened in FIFO order. This is exemplified in the following scenarios.

- Threads waiting to enter synchronized blocks are granted access to the synchronized block in priority order.
- A blocked thread that becomes ready to run is given access to a processor in priority order.

- A thread whose priority is explicitly set by itself or another thread is given access to a processor in priority order.
- A thread that performs a yield will be given access to the processor after waiting for threads of the same priority to be given a processor.
- Threads that are preempted in favor of a thread with higher priority may be given access to a processor at any time as determined by a particular implementation. The implementation is required to provide documentation stating exactly the algorithm used for granting such access.

In any case, there needs to be a fixed upper bound on the time required to enter a synchronized block for an unlocked monitor.

2.2.3 Execution Eligibility

Since an implementation of the `RTSJ` may provide schedulers other than priority-based schedulers, the notion of priority can be generalized to execution eligibility. Execution eligibility defines a partial ordering over all tasks for determining which task should run before which other tasks. Execution eligibility may be determined dynamically. For example, earliest deadline first (EDF) scheduling determines execution eligibility ordering by the order of the next deadlines for each of its tasks. The notion of priority, as described above, can be generalized to execution eligibility to integrate other schedulers into an `RTSJ` implementation.

2.2.4 Wait-Free Queues

While the `RTSJ` requires that the execution of schedulables which do not access the heap must not be delayed by garbage collection on behalf of lower-priority schedulables, an application can cause such a schedulable to wait for garbage collection by synchronizing using an object shared with a heap-using thread or schedulable. The `RTSJ` provides wait-free queue classes to provide protected, non-blocking, shared access to objects accessed by both regular Java threads and schedulables, which do not access the heap.

2.3 Asynchrony

Since a realtime system must be able to react to the outside world, the system needs to be able to change its execution flow asynchronously to the current execution. All external signals, whether interrupts, messages, or timed events, are asynchronous with respect to ongoing computation. This means that computation must be both startable and stoppable based on external stimuli.

2.3.1 Asynchronous Events

Asynchronous event provide a means of starting computation based on external stimuli. The asynchronous event facility is based on two classes: `AbstractAsyncEvent` and `AbstractAsyncEventHandler`. An `AbstractAsyncEvent` object represents something that can happen, like a POSIX signal, a hardware interrupt, or a computed event like an airplane entering a specified region. When one of these events occurs, which is indicated by the `fire()` method being called, the associated instances of `AbstractAsyncEventHandler` are scheduled and the `handleAsyncEvent()` methods are invoked, thus the required logic is performed. Also, methods on `AbstractAsyncEvent` are provided to manage the set of instances of `AbstractAsyncEventHandler` associated with the instance of `AbstractAsyncEvent`.

An instance of an `AbstractAsyncEventHandler` can be thought of as something similar to a thread. When an event fires, the associated handlers are scheduled and the `handleAsyncEvent()` methods are invoked. What distinguishes an `AbstractAsyncEventHandler` from a simple `Runnable` is that an `AbstractAsyncEventHandler` has associated instances of `ReleaseParameters`, `SchedulingParameters` and `MemoryParameters` that control the actual execution of the handler once the associated `AbstractAsyncEvent` is fired. When an event is fired, the handlers are executed asynchronously, scheduled according to the associated `ReleaseParameters` and `SchedulingParameters` objects, in a manner that looks like the handler has just been assigned to its own thread. It is intended that the system can cope well with situations where there are large numbers of instances of `AbstractAsyncEvent` and `AbstractAsyncEventHandler` (tens of thousands), since the number of fired (in progress) handlers is expected to be much smaller.

There are specialized forms of `AbstractAsyncEvent`: `AsyncEvent`, `AsyncLongEvent`, and `AsyncObjectEvent` for events that are stateless, carry a `long` payload, and carry an `Object` payload, respectively. They are matched by specialized forms of `AbstractAsyncEventHandler`: `AsyncEventHandler`, `AsyncLongEventHandler`, and `AsyncObjectEventHandler`. Most external events are stateless, but sometimes it is helpful to be able to receive some information about the event or pass some data with the event. The `Long` and `Object` variants enable this and the `POSIXRealtimeSignal` takes advantage of it.

Another specialized form of an `AsyncEvent` is the `Timer` class, which represents an event whose occurrence is driven by time. There are two forms of Timers: the `OneShotTimer` and the `PeriodicTimer`. Instances of `OneShotTimer` fire once, at the specified time. Periodic timers fire initially at the specified time, and then periodically according to a specified interval.

Timers are driven by `Clock` objects. There is a special `Clock` object, `Clock.getRealtimeClock()`, that represents the realtime clock. The Clock class may be extended to represent other clocks, which the underlying system might make available (such as an execution-time clock of some granularity).

2.3.2 Asynchronous Transfer of Control

Many event-driven computer systems that tightly interact with external physical systems (e.g., humans, machines, control processes, etc.) may require mode changes in their computational behavior as a result of significant changes in the non-computer real-world system. It simplifies the architecture of a system when a task can be programmatically terminated when an external physical system change causes its computation to be superfluous. Without this facility, a thread or set of threads have to be coded so that their computational behavior anticipates all of the possible transitions among possible states of the external system. When the external system makes a state transition, the changes in computation behavior can be managed by an oracle that terminates a set of threads required for the old state of the external system, and invokes a new set of threads appropriate for the new state of the external system. Since the possible state transitions of the external system are encoded in only the oracle and not in each thread, the overall system design is simpler.

There is a second requirement for a mechanism to terminate some computation, where a potentially unbounded computation needs to be done in a bounded period of time. In this case, if that computation can be executed with an algorithm that is iterative, and produces successively refined results, the system could abandon the computation early and still have usable results. The `RTSJ` supports aborting a computation by signalling from another thread, or the passage of time, with a feature termed Asynchronous Transfer of Control (ATC).

An example of the second case is processing compressed video for a human controller. The system knows that a new frame must be produced at a constant update frequency. The cost of each iteration is highly variable and the minimum required latency to terminate the computation and receive the last consistent result is much less than the mean iteration cost and bound. Therefore, using ATC for interrupting a computation to capture an intermediate result at the expiration of a known time bound is a convenient programming style. Of course, there are other kinds of programming tasks that may also benefit from ATC.

2.3.3 Principles

The `RTSJ`'s approach to ATC uses asynchronous interruptions and exceptions, and is based on several guiding principles covering methodology, expressiveness, semantics, and pragmatic concerns.

2.3.3.1 Methodological Principles

- A method must explicitly indicate its susceptibility to ATC, i.e., it is asynchronously interruptible. Since legacy code or library methods might have been written assuming no ATC, by default ATC must be turned off (more precisely, must be deferred as long as control is in such code).
- Even if a method allows ATC, some code sections must be executed to completion and thus ATC is deferred in such sections. These ATC-deferred sections are synchronized methods, static initializers, and synchronized statements.
- Code that responds to an ATC does not return to the point in the schedulable where the ATC was triggered; that is, an ATC is an unconditional transfer of control. Resumptive semantics, which returns control from the handler to the point of interruption, are not needed since they can be achieved through other mechanisms (in particular, an `AsyncEventHandler`).

2.3.3.2 Expressibility Principles

- A mechanism is needed through which an ATC can be explicitly triggered in a target schedulable. This triggering may be direct (from a source thread or schedulable) or indirect (through an asynchronously interrupted exception).
- It must be possible to trigger an ATC based on any asynchronous event including an external happening or an explicit event firing from another thread or schedulable. In particular, it must be possible to base an ATC on a timer going off.
- Through ATC it must be possible to abort a realtime thread but in a manner that does not carry the dangers of the `Thread` class's `stop()` and `destroy()` methods.

2.3.3.3 Semantic Principles

- If ATC is modeled by exception handling, there must be some way to ensure that an asynchronous exception is only caught by the intended handler and not, for example, by an all-purpose handler that happens to be on the propagation path.
- Nested ATCs must work properly. For example, consider two, nested ATC-based timers and assume that the outer timer has a shorter time-out than the nested, inner timer. If the outer timer times out while control is in the nested code of the inner timer, then the nested code must be aborted (as soon as it is outside an ATC-deferred section), and control must then transfer to the appropriate `catch` clause for the outer timer. An implementation that either handles the outer time-out in the nested code, or that waits for the longer (nested) timer, is incorrect.

2.3.3.4 Pragmatic Principles

- There should be straightforward idioms for common cases such as timer handlers and realtime thread termination.
- If code with a time-out completes before the timer's expiration, the timer needs to be automatically stopped and its resources returned to the system.

2.3.4 Asynchronous Realtime Thread Termination

A special case of stopping a particular computation is stopping a thread. Earlier versions of the Java language supplied mechanisms for achieving these effects: in particular the methods `stop()` and `destroy()` in class `Thread`. However, since `stop()` could leave shared objects in an inconsistent state, `stop()` has been deprecated. The use of `destroy()` can lead to deadlock (if a thread is destroyed while it is holding a lock) and although it was not deprecated until version 1.5 of the Java specification, its usage has long been discouraged. A goal of the `RTSJ` was to meet the requirements of asynchronous thread termination without introducing the dangers of the `stop()` or `destroy()` methods.

The `RTSJ` accommodates safe asynchronous realtime thread termination through a combination of the asynchronous event handling and the asynchronous transfer of control mechanisms. To create such a set of realtime threads consider the following steps:

- make all of the application methods of the realtime thread asynchronously interruptible;
- create an oracle which monitors the external world by setting up an asynchronous event with a number of asynchronous event handlers, which is fired when an appropriate mode change;
- have the handlers call `interrupt()` on each of the realtime threads affected by the change; then
- after the handlers call `interrupt()`, have them create a new set of realtime threads appropriate to the current state of the external world.

The effect of the event is to cause each interruptible method to abort abnormally by transferring control to the appropriate catch clause. Ultimately the `run()` method of the realtime thread will complete normally.

This idiom provides a quick (if coded to be so) but orderly clean up and termination of the realtime thread. Note that the oracle can comprise as many or as few asynchronous event handlers as appropriate.

2.4 Clocks, Time, and Timers

Realtime systems require a high resolution notion of time. Both very small units and very long periods of time must be uniformly representable, a range that is not even representable with a `long` value. Furthermore, a time can represent an absolute value, usually represented as some absolute fixed point in time plus an offset, or it can represent an interval of time. The time classes defined in Chapter 9 support a longs worth of seconds and another integer for nanoseconds.

2.5 Memory Management

The Java language is designed around automatic memory management, in particular garbage collection. Unfortunately, though garbage collection is a functional safety and security feature, conventional garbage collectors interrupt the normal flow of control in a program. Therefore, garbage-collected memory heaps had been considered an obstacle to realtime programming due to the potential for unpredictable latencies introduced by the garbage collector. Though convention collectors still have these drawbacks, there are now realtime collectors that can be used for hard realtime application. Still, the `RTSJ` provides an alternative to garbage collection for systems which require it, either because they do not have a garbage collector or deterministic garbage collector, or require heap partitioning for some other reason. Extensions to the memory model, which support memory management in a manner that does not interfere with the ability of realtime code to provide deterministic behavior, are provided to support these alternatives. This goal is accomplished by providing memory areas for the allocation of objects outside of the garbage-collected heap for both short-lived and long-lived objects. In order to provide additional separation between the garbage collector and schedulables which do not require its services, a schedulable can be marked no-heap to indicate that it never accesses the heap.

2.5.1 Memory Areas

The `RTSJ` introduces the concept of a memory area. A memory area represents an area of memory that may be used for allocating objects. Some memory areas exist outside of the heap and place restrictions on what the system and garbage collector may do with objects allocated within. Objects in some memory areas are never garbage collected; however, the garbage collector must be capable of scanning these memory areas for references to any object within the heap to preserve the integrity of the heap.

There are four basic types of memory areas:

- Heap memory represents an area of memory that is the heap. The `RTSJ` does not change the determinant of lifetime of objects on the heap. The lifetime is still determined by visibility.
- Immortal memory represents an area of memory containing objects that may be referenced without exception or garbage collection delay by any schedulable, specifically including no-heap realtime threads and no-heap asynchronous event handlers.
- Scoped memory provides a mechanism for managing objects that have a lifetime defined by their scope. It is akin to, but more general than, allocating objects on the thread stack.
- Physical memory allows objects to be created within specific physical memory regions that have particular important characteristics, such as memory that has substantially faster access.

2.5.2 Heap Memory

Heap memory is the memory area used by Java by default. It is garbage collected and the access time to objects in this area are not guaranteed unless the implementation supports realtime garbage collection. The `RTSJ`, as with conventional Java, supports only one Heap in a system. Multiple heaps are only practical in one of two configurations: the heaps are completely independent of one another or there are subsidiary heaps from which a program may not store references in the main heap. In other words, the subsidiary heaps can reference the main heap but not vise versa. Currently, the `RTSJ` does not address these cases.

2.5.3 Immortal Memory

`ImmortalMemory` is a memory resource shared among all schedulable objects and threads in an application. Objects allocated in `ImmortalMemory` are always available to non-heap threads and asynchronous event handlers without the possibility of a delay for garbage collection.

2.5.4 Scoped Memory

The `RTSJ` introduces the concept of scoped memory. A memory scope is used to give bounds to the lifetime of any objects allocated within it. When a scope is entered, every use of `new` causes the memory to be allocated from the active memory scope. A scope may be entered explicitly, or it can be attached to a schedulable which will effectively enter the scope before it executes the object's `run()` method.

The contents of a scoped memory are discarded when no object in the scope can be accessed. This is done by a technique similar to reference counting the scope.

A conforming implementation might maintain a count of the number of external references to each memory area. The reference count for a `ScopedMemory` area would be increased by entering a new scope through the `enter()` method of `MemoryArea`, by the creation of a schedulable using the particular `ScopedMemory` area, or by the opening of an inner scope. The reference count for a `ScopedMemory` area would be decreased when returning from the `enter()` method, when the schedulable using the `ScopedMemory` terminates, or when an inner scope returns from its `enter()` method. When the count drops to zero, the finalize method for each object in the memory would be executed to completion. Reuse of the scope is blocked until finalization is complete.

Scopes may be nested. When a nested scope is entered, all subsequent allocations are taken from the memory associated with the new scope. When the nested scope is exited, the previous scope is restored and subsequent allocations are again taken from that scope.

Because of the lifetimes of scoped objects, it is necessary to limit the references to scoped objects, by means of a restricted set of assignment rules. A reference to a scoped object cannot be assigned to a variable from an outer scope, or to a field of an object in either the heap or the immortal area. A reference to a scoped object may only be assigned into the same scope or into an inner scope. The virtual machine must detect illegal assignment attempts and must throw an appropriate exception when they occur.

The flexibility provided in choice of scoped memory types allows the application to use a memory area that has characteristics that are appropriate to a particular syntactically defined region of the code.

2.5.5 Physical Memory Areas

In many cases, systems needing the predictable execution of the `RTSJ` will also need to access various kinds of memory at particular addresses for performance or other reasons. Consider a system in which very fast static RAM was programmatically available. A design that could optimize performance might wish to place various frequently used Java objects in the fast static RAM. The `LTPhysicalMemory` and `ImmortalPhysicalMemory` classes provide the programmer this flexibility. The programmer would construct a physical memory object on the memory addresses occupied by the fast RAM.

2.5.6 Budgeted Allocation

The `RTSJ` also provides limited support for providing memory allocation budgets for schedulables using memory areas. Maximum memory area consumption and maximum allocation rates for individual schedulable objects may be specified when they are created.

2.6 Device Access and Raw Memory

The `RTSJ` defines classes for programmers wishing to directly access physical memory from code written in the Java language. The `RawMemory<Size>` types define methods that enable the programmer to construct an object that represents a range of physical addresses, where the `Size` represents a word size, i.e., byte, short, int, long, float, and double. Access to the physical memory is then accomplished through `get<Size>()` and `set<Size>()` methods of that object. No semantics other than the `set<Size>()` and `get<Size>()` methods are implied. One the other hand, the `LTPhysicalMemory` and `ImmortalPhysicalMemory` classes enable programmers to construct an object that represents a range of physical memory addresses. When this object is used as a `MemoryArea` other objects can be constructed in the physical memory using the new keyword as appropriate. Factories can be used to create the desired type of both physical and raw memory.

2.6.1 Raw Memory Access

An instance of `RawMemoryType` models a range of physical memory locations as a fixed sequence of elements of a given size. The elements correspond to Java primitive types. For objects that access more than a single physical address, elements can be accessed through offsets from the base, where the offset is measured in multiples of the element size, not necessarily the byte offset in memory.

The `RawMemoryType` interface enables a realtime program to implement device drivers, memory-mapped registers, I/O space mapped registers, flash memory, battery-backed RAM, and similar low-level hardware.

A raw memory area cannot contain references to Java objects. Such a capability would be unsafe (since it could be used to defeat Java's type checking) and error prone (since it is sensitive to the specific representational choices made by the Java compiler).

2.7 System Options

POSIX defines some convenient interfaces for interacting with the system. These interactions include catching keyboard interrupts, user-to-process signaling, and interprocess signaling. Many realtime operating systems support this POSIX signal interface. For this reason, the `RTSJ` provides a POSIX signal interface. Though many of the features POSIX signals provide are also available on most other operating systems, the specification does not require the POSIX signal interface to be emulated on these other platforms. Thus they are optional in the sense that they are only required on systems that directly support POSIX signals.

2.8 Exceptions

Aside from several new exceptions, the `RTSJ` provides a new interface for using exceptions without creating ephemeral objects and some new treatment of exceptions surrounding asynchronous transfer of control.

Using exceptions is resource intensive, since a new exception is allocated for each throw. This is particularly a problem for scoped memory, since scopes may need to be sized much larger than otherwise necessary to hold exceptions and their stack traces. Additionally, the information they contain cannot be propagated beyond the scope in which they are allocated. To better support scoped, immortal, and physical memory, a new class of throwable has been included: PreallocatedThrowable. Exceptions and Errors which implement this interface are not thrown in the usual manner, but with a style that does not require memory to be allocated at all.

Asynchronous transfer of control can cause the exception that triggered it to be propagated even when it is caught but the underlying interrupt is not cleared. The system rethrows the exception once the catch is finished. This is necessary since the exception hierarchy is poorly designed. There is no common base class only for all checked exceptions, so use code often contains a catch for `Exception` when only checked exceptions need to be caught. Even the JVM specification wording is awkward on this point.

2.9 Summary

The `RTSJ` refines the semantics of threads, scheduling, synchronization, memory management, and exceptions and adds features to support realtime threads, realtime scheduling, configuring synchronization, asynchrony, representing time, clocks and timers, additional methods for memory management, device access and raw memory, system options. These features and semantic refinements to the Java language and virtual machine have been outlined above, but the description does not constitute a definition for them. In other words, it is not normative. The normative chapters follow.

Chapter 3

Requirements and Conventions

This specification is a contract between the specification implementer and the user who writes a program to run on an implementation. To be able to support both implementation and use, many chapters provide additional rationale to help both the implementer and the user understand the intention behind the normative text. The remainder of this specification, including this chapter, is normative, except for the introductory text in each chapter and the sections named Rationale.

3.1 Base Requirements

The base requirements of this specification are as follows.

- Except as specifically required by this specification, any implementation shall fully conform to a Java platform configuration.
- Any implementation of this specification shall implement all classes and methods in the base module of this specification.
- Except as noted in this chapter, all classes and methods in an implemented module shall be implemented.
- The `javax.realtime` package shall contain no public or protected classes or methods not included in this specification.
- A realtime JVM implementation shall not be implemented in a way that permits unbounded priority inversion in any scheduling interaction it implements.
- Subject to the usual assumptions, the methods in `javax.realtime` can safely be used concurrently by multiple threads unless otherwise documented.
- Static final values, as found in `AperiodicParameters`, `PhysicalMemoryManager`, `SporadicParameters`, `RealtimeSystem`, and `PriorityScheduler`, shall be implemented such that their values cannot be resolved by a conformant Java compiler (Java source to byte code).

Many aspects of this specification set a minimum requirement, but permit the implementation latitude in its implementation. For instance, the required priority scheduler requires at least 28 consecutively numbered realtime priorities. It does not, however, specify the numeric values of the maximum and minimum realtime priorities. Implementations are encouraged to offer as many realtime priority levels immediately above the conventional Java priorities as they can support.

Except where otherwise specified, when this specification requires object creation the object is created in the current allocation context.

3.2 Modules

The original `RTSJ` specification was conceived, with the exception of some optional features, as a monolith specification. This has inhibited the adoption of the `RTSJ` beyond the hard realtime community, because some of the features were considered to have an overly negative impact on overall JVM performance. Version 2.0 addresses this by breaking the specification into modules.

Modules provide a means of grouping like functionality together in a way that promotes maximal adoption for various implementation classes. A conventional JVM could simply implement the Base Module, without providing any realtime guarantees at all, to provide programmers with the benefits of features such as asynchronous event programming as an alternative to conventional threading. A hard realtime implementation could implement all modules to provide the maximal flexibility and functionality to the realtime programmer. Both would benefit from easier migration of code to realtime systems.

Every `RTSJ` implementation shall provide the Base Module functionality, but all other modules are optional. The optional modules are the Device Module and the Alternative Memory Areas module. In addition, there are a couple of optional features as well. This give the implementation some choice over which modules and features to include and which not.

3.2.1 Base Module

The Base Module adds the concepts of processor affinity, threads with realtime scheduling, and asynchronous event handling. This includes the notion of executing code at a given time interval, providing a much more stable response than using `sleep` in a loop. These features should have no impact on the overall performance of a system that implements them, but enrich the programming modules available to the programmer. The classes and interfaces required in this module are listed below.

- AbsoluteTime (Section 9.4.1.1)

- AbstractAsyncEventHandler[1] (Section 8.4.3.2)
- AbstractAsyncEvent (Section 8.4.3.1)
- ActiveEventDispatcher (Section 8.4.3.3)
- ActiveEvent (Section 8.4.1.1)
- Affinity (Section 6.4.2.1)
- AperiodicParameters (Section 6.4.2.2)
- ArrivalTimeQueueOverflowException (Section 14.2.2.1)
- AsyncEventHandler (Section 8.4.3.5)
- AsyncEvent (Section 8.4.3.4)
- AsyncLongEventHandler (Section 8.4.3.7)
- AsyncLongEvent (Section 8.4.3.6)
- AsyncObjectEventHandler (Section 8.4.3.9)
- AsyncObjectEvent (Section 8.4.3.8)
- BoundAbstractAsyncEventHandler (Section 8.4.1.2)
- BoundAsyncEventHandler (Section 8.4.3.10)
- BoundAsyncLongEventHandler (Section 8.4.3.11)
- BoundAsyncObjectEventHandler (Section 8.4.3.12)
- CeilingViolationException (Section 14.2.2.2)
- Clock (Section 10.4.2.1)
- ConfigurationParameters (Section 5.3.2.1)
- DeregistrationException (Section 14.2.2.3)
- DuplicateFilterException (Section 14.2.2.5)
- GarbageCollector (Section 13.3.1.1)
- HeapMemory (Section 11.4.3.1)
- HighResolutionTime (Section 9.4.1.2)
- ImportanceParameters (Section 6.4.2.3)
- LateStartException (Section 14.2.2.7)
- MemoryArea (Section 11.4.3.6)
- MonitorControl (Section 7.3.1.1)
- OneShotTimer (Section 10.4.2.2)
- PeriodicParameters (Section 6.4.2.4)
- PeriodicTimer (Section 10.4.2.3)
- PhasingPolicy (Section 5.3.1.1)
- PriorityInheritance (Section 7.3.1.3)
- PriorityParameters (Section 6.4.2.5)
- PriorityScheduler (Section 6.4.2.6)
- ProcessingGroupParameters (Section 6.4.2.7)
- ProcessorAffinityException (Section 14.2.2.13)
- RealtimeSecurity (Section 13.3.1.6)

[1]The no-heap flag is present, but can only be set if the Memory Module is supported.

- RealtimeSystem (Section 13.3.1.7)
- RealtimeThread (Section 5.3.2.2)
- RegistrationException (Section 14.2.2.14)
- RelativeTime (Section 9.4.1.3)
- ReleaseParameters (Section 6.4.2.8)
- ResourceLimitError (Section 14.2.3.5)
- Schedulable (Section 6.4.1.2)
- Scheduler (Section 6.4.2.9)
- SchedulingParameters (Section 6.4.2.10)
- SporadicParameters (Section 6.4.2.11)
- Timable (Section 10.4.1.3)
- TimeDispatcher (Section 10.4.2.4)
- Timer (Section 10.4.2.5)

3.2.2 Device Module

The Device Module provides a low level interface for interacting with the real world. Though realtime control systems need this kind of interaction, other systems can benefit from it as well. Data collection, that is not time critical is a good example. For instance, monitoring the temperature or humidity in a room could be done easily with off-the-self hardware using this module. The classes required in this module are listed below.

- Happening (Section 12.4.3.1)
- HappeningDispatcher (Section 12.4.3.2)
- POSIXRealtimeSignal (Section 13.3.1.2)
- POSIXRealtimeSignalDispatcher (Section 13.3.1.3)
- POSIXSignal (Section 13.3.1.4)
- POSIXSignalDispatcher (Section 13.3.1.5)
- RawBufferFactory (Section 12.4.3.4)
- RawMemory (Section 12.4.1.16)
- RawMemoryFactory (Section 12.4.3.5)
- RawMemoryRegion (Section 12.4.3.6)
- RawMemoryRegionFactory (Section 12.4.1.17)
- UnsupportedRawMemoryRegionException (Section 12.4.2.1)
- RawByte (Section 12.4.1.1)
- RawByteReader (Section 12.4.1.2)
- RawByteWriter (Section 12.4.1.3)
- RawDouble (Section 12.4.1.4)
- RawDoubleReader (Section 12.4.1.5)
- RawDoubleWriter (Section 12.4.1.6)
- RawFloat (Section 12.4.1.7)

- RawFloatReader (Section 12.4.1.8)
- RawFloatWriter (Section 12.4.1.9)
- RawInt (Section 12.4.1.10)
- RawIntReader (Section 12.4.1.11)
- RawIntWriter (Section 12.4.1.12)
- RawLong (Section 12.4.1.13)
- RawLongReader (Section 12.4.1.14)
- RawLongWriter (Section 12.4.1.15)
- RawMemoryFactory (Section 12.4.3.5)
- RawMemoryRegionFactory (Section 12.4.1.17)
- RawShort (Section 12.4.1.18)
- RawShortReader (Section 12.4.1.19)
- RawShortWriter (Section 12.4.1.20)

3.2.3 Alternative Memory Areas Module

The Alternative Memory Areas Module provides an alternative to a single heap with garbage collection model for memory management. Most of the facilities are centered around providing an alternative to garbage collection, but facilities for providing what memory to use for Java objects is also addressed. The classes required in this module are listed below.

- AlignmentError (Section 14.2.3.1)
- ChildScopeVisitor (Section 11.4.1.1)
- IllegalAssignmentError (Section 14.2.3.3)
- ImmortalMemory (Section 11.4.3.2)
- ImmortalPhysicalMemory (Section 11.4.3.3)
- InaccessibleAreaException (Section 14.2.2.6)
- LTMemory (Section 11.4.3.4)
- LTPhysicalMemory (Section 11.4.3.5)
- MemoryAccessError (Section 14.2.3.4)
- MemoryInUseException (Section 14.2.2.9)
- MemoryParameters (Section 11.4.3.7)
- MemoryScopeException (Section 14.2.2.10)
- MemoryTypeConflictException (Section 14.2.2.11)
- MITViolationException (Section 14.2.2.8)
- NewPhysicalMemoryManager (Section 11.4.3.8)
- NoHeapRealtimeThread (Section 15.3.2.1)
- OffsetOutOfBoundsException (Section 14.2.2.12)
- PhysicalMemoryCharacteristic (Section 11.4.1.2)
- PhysicalMemoryFilter (Section 11.4.1.3)
- PhysicalMemoryManager (Section 15.3.2.3)

- PhysicalMemoryModule (Section 11.4.3.9)
- PhysicalMemoryName (Section 15.3.1.1)
- PhysicalMemoryTypeFilter (Section 15.3.1.2)
- PinnableMemory (Section 11.4.3.10)
- ScopedCycleException (Section 14.2.2.15)
- ScopedMemory (Section 11.4.3.11)
- SizeEstimator (Section 11.4.3.12)
- SizeOutOfBoundsException (Section 14.2.2.16)
- StackedMemory (Section 11.4.3.13)
- ThrowBoundaryError (Section 14.2.3.6)
- UnsupportedPhysicalMemoryException (Section 14.2.2.18)
- VirtualMemoryCharacteristic (Section 11.4.1.4)
- WaitFreeReadQueue (Section 7.3.1.5)
- WaitFreeWriteQueue (Section 7.3.1.6)

3.2.4 Optional Features

Even with modules it is difficult to elliminate all optional features. These features are either not easy to implement on all platforms or have the potential to cause a significant performance overhead. Therefore, an application cannot depend on them to be present in every implementation. However, if an optional facility is implemented, the application may rely on it to behave as specified here. Those extensions are illustrated in Table 3.1.

Table 3.1: `RTSJ` Options

Cost enforcement	Enables the application to control the processor utilization of a schedulable.
Processing Group enforcement	Enables the application to control the processor utilization of a group of schedulables
Processing Group deadline less than period	Enables the application to specify a processing group deadline less than the processing group period
Allocation-rate enforcement on heap allocation	Enables the application to limit the rate at which a schedulable creates objects in the heap.
Interrupt Service Routine	Provides first level interrupt processing in Java.
Asynchronous Transfer of Control (ATC)	Enables schedules to be interrupted asynchronously.

The `ProcessingGroupParameters` class is only functional on systems that support the processing group enforcement option. Cost enforcement, and cost overruns handlers are only functional on systems that support the cost enforcement option. If processing group enforcement is supported, `ProcessingGroupParameters` shall function as specified. If cost enforcement is supported, cost enforcement, and cost overrun handlers shall function as specified.

In implementations where the processing group deadline less than period is not supported, values passed to the constructor for `ProcessingGroupParameters` and its `setDeadline` method are constrained to be equal to the period. If the option is supported, processing group deadlines less than the period shall be supported and function as specified.

In implementations where heap allocation rate enforcement is supported, it shall be implemented as specified. If heap allocation rate enforcement is not supported, the allocation rate attribute of `MemoryParameters` shall be checked for validity but otherwise ignored by the implementation.

First level interrupt handling can only be supported in certain contexts, such as in kernel space and in a device driver context in user space on systems that support this feature. Normally user space programs cannot handle interrupts directly. The class should be present in every system that implements the device module, but in implementations that do not support first level interrupt handling, the register should always throw an `UnsupportedOperationException`.

ATC, if implemented, requires the following classes:

- Timed (Section 8.4.2.2)
- AsynchronouslyInterruptedException (Section 8.4.2.1)
- Interruptible (Section 8.4.1.3)
- InterruptServiceRoutine (Section 12.4.3.3)

Extensions to this specification are allowed, but shall not require changes to the `javax.realtime` package tree.

3.2.5 Deprecated Classes

Classes that have been deprecated as of this specification are not part of any module, but may be implemented by a full `RTSJ` implementation. They comprise the following class:

- PhysicalMemoryManager (Section 15.3.2.3)
- PhysicalMemoryName (Section 15.3.1.1)
- PhysicalMemoryTypeFilter (Section 15.3.1.2)
- POSIXSignalHandler (Section 15.3.2.2)
- RationalTime (Section 15.3.2.4)
- RawMemoryAccess (Section 15.3.2.5)
- RawMemoryFloatAccess (Section 15.3.2.6)
- VTMemory (Section 15.3.2.7)
- VTPhysicalMemory (Section 15.3.2.8)

- WaitFreeDequeue (Section 7.3.1.4)

They are documented fully in Chapter 15.

3.3 Conditionally-Required Facilities

An implementation shall support conditionally-required facilities if the underlying hardware and software permits. This specification includes two conditionally-required facilities:

`POSIXSignal`	This class shall be implemented on every platform where POSIX signals are supported.
`RawMemory`	If the system supports address translation, e.g., has an MMU, and implements the Device Module, the implementation shall support the memory mapping features of the raw memory access classes.

3.3.1 Options for Development Platforms

The following semantics are optional for an `RTSJ` implementation designed and licensed exclusively as a development tool.

- The priority scheduler need not support fixed-priority preemptive scheduling or the priority inversion avoidance algorithms. This does not excuse an implementation from fully supporting the relevant APIs. It only reduces the required behavior of the underlying scheduler to the level of the scheduler in the Java specification extended to at least 28 priorities.
- No semantics constraining timing beyond the requirements of the Java specifications need be supported. Specifically, garbage collection may delay any thread without bound and any delay in delivering asynchronously interrupted exceptions is permissible including never delivering the exception. Note, however, that if any AIE other than the generic AIE is delivered, it shall meet the AIE semantics, and all heap-memory-related semantics other than preemption remain fully in effect. Further, relaxed timing does not imply relaxed sequencing. For instance, semantics for scoped memory shall be fully implemented.
- The `RTSJ` semantics that alter standard Java method behavior, such as the modified semantics for `Thread.setPriority` and `Thread.interrupt`, are not required for a development tool, but such deviations from the `RTSJ` shall be documented, and the implementation shall be able to generate a runtime warning each time one of these methods deviates from standard `RTSJ` behavior.

These relaxed requirements set a floor for `RTSJ` development system tool implementations. A development tool may choose to implement semantics that are not required.

3.4 Required Documentation

Each implementation of the `RTSJ` is required to provide documentation for several behaviors.

- If schedulers other than the base priority scheduler are available to applications, the behavior of the scheduler and its interaction with each other scheduler as detailed in the Scheduling chapter shall be documented. The list of classes that constitute schedulable objects for the scheduler, unless that list is the same as the list of schedulables for the base scheduler, shall be included. If there are restrictions on use of the scheduler from a non-heap context, these restrictions shall be documented as well.
- A schedulable that is preempted by a higher-priority schedulable is placed in the queue for its active priority, at a position determined by the implementation. If the preempted schedulable is not placed at the front of the appropriate queue the implementation shall document the algorithm used for such placement. Placement at the front of the queue may be required in a future version of this specification.
- An implementation is required to document the granularity at which the current CPU consumption is updated for cost monitoring and cost enforcement, when the later is implemented.
- The implementation shall fully document the behavior of any subclasses of `GarbageCollector`.
- An implementation that provides any `MonitorControl` subclasses not detailed in this specification shall document their effects, particularly with respect to priority inversion control and which (if any) schedulers fail to support the new policy.
- If on losing "boosted" priority due to a priority inversion avoidance algorithm, the schedulable is not placed at the front of its new queue, the implementation shall document the queuing behavior.
- For any available scheduler other than the base scheduler, an implementation shall document how, if at all, the semantics of synchronization differ from the rules defined for the default `PriorityInheritance` monitor control policy. It shall supply documentation for the behavior of the new scheduler with priority inheritance (and, if it is supported, priority ceiling emulation protocol) equivalent to the semantics for the base priority scheduler found in the Synchronization chapter. If there are restrictions on use of the scheduler from a no-heap context, the documentation shall detail the effect of these restrictions for each `RTSJ` API.

- The worst-case response interval between firing an `AsyncEvent` because of a bound happening to releasing an associated `AsyncEventHandler` (assuming no higher-priority schedulables are runnable) shall be documented for some reference architecture.
- The interval between firing an `AsynchronouslyInterruptedException` at an ATC-enabled thread and first delivery of that exception (assuming no higher-priority schedulables are runnable) shall be documented for some reference architecture.
- If cost enforcement is supported, and the implementation assigns the cost of running finalizers for objects in scoped memory to any schedulable other than the one that caused the scope's reference count to drop to zero by leaving the scope, the rules for assigning the cost shall be documented.
- If cost enforcement is supported, and enforcement (blocked-by-cost-overrun) can be delayed beyond the enforcement time granularity, the maximum such delay shall be documented.
- If the implementation of `RealtimeSecurity` is more restrictive than the required implementation, or has run-time configuration options, those features shall be documented.
- For each supported clock, the documentation shall specify whether the resolution is settable, and if it is settable the documentation shall indicate the supported values.
- If an implementation includes any clocks other than the required realtime clock, their documentation shall indicate in what contexts those clocks can be used. If they cannot be used in no-heap context, the documentation shall detail the consequences of passing the clock, or a time that uses the clock to a no-heap schedulable.

3.5 Conventions

Throughout the `RTSJ`, when we use the word *code*, we mean code written in the Java programming language. When we mention the Java language in the `RTSJ`, that also refers to the Java programming language. The use of the term *heap* in the `RTSJ` will refer to the heap used by the runtime of the Java language.

3.6 Definitions

A *thread* is an instance of the `java.lang.Thread` class.
A *realtime thread* is an instance of the `javax.realtime.RealtimeThread` class.
A *Java thread* is a thread that is not a realtime thread.

A *no-heap realtime thread* is an instance of the `javax.realtime.RealtimeThread` class that will not access the heap.
An *event handler* is an instance of the `javax.realtime.AbstractAsyncEventHandler` class.
The term *schedulable* refers to any object that is of type `Schedulable`, and is recognized as a dispatchable entity by the base scheduler. The base scheduler's set of schedulables comprises instances of `RealtimeThread` and `AsyncEventHandler`. Other schedulers may support a different set of schedulables, but this specification only defines the behavior of the base scheduler so the term schedulable should be understood as "schedulable by the base scheduler."

Chapter 4

Conventional Java Classes and Language

Though compatibility with conventional Java, i.e., any Java runtime environments that implement the Java Virtual Machine Specification and the Java Language Specification but not the RTSJ, is the first concern of this specification, there are several several cases where being able to meet realtime constraints requires a tightening of the semantics of the virtual machine and some subtle changes to the semantics of two key classes: `java.lang.Thread` and `java.lang.ThreadGroup`. These constraints and changes place additional requirements on scheduling, the memory model, and memory management. Finally, the specification defines a new type of concurrent activity called an asynchronous event handler; hence, the meaning of current thread has a different interpretation than in standard java.

4.1 Scheduling

How threads are scheduled in a realtime system is quite different than what one expects in a conventional Java virtual machine. For compatibility, this means that there must be a domain where conventional Java threads are scheduled in a familiar way and another domain that supports realtime scheduling. This separation is done in part via thread priority.

Threads running with the conventional ten priorities defined in Java should be scheduled as expected. Unfortunately, in order to ease the porting of Java to different environments, the scheduling of conventional Java threads is under specified. This has been resolved to avoid surprising the programmer in practice by providing some sort of fair scheduling for these threads. For threads running in these priorities an implementation of this specification should provide some notion of fair scheduling between threads with priority between one and ten inclusive.

Realtime threads need a stronger notion of prioritization than conventional Java threads, so this specification requires the implementation of at least a priority-preemptive scheduler with run to completion (or next suspension point) semantics. Priorities above the conventional ten priorities are used for this. Multithreaded code that runs with the priority-preemptive scheduler (or any other realtime scheduler) is more prone to deadlock or starvation than code run with fair scheduling. The changes to `Thread` and `ThreadGroup` are to support this realtime scheduling.

- The semantics of set and get methods for priority in `Thread` differ for realtime threads.
- The `ThreadGroup` class's behavior differs with respect to realtime threads.
- The behavior of the `ThreadGroup`-related methods in `Thread` differ when they are applied to realtime threads.

Code running at realtime priorities can also block conventional Java threads, possibly indefinitely.

4.1.1 Priority

The methods `setPriority` and `getPriority` in `java.lang.Thread` are `final`. The realtime thread classes are consequently not able to override them and modify their behavior to suit the requirements of the `RTSJ` scheduler. To bring the `java.lang.Thread` class in line with its realtime subclasses, the semantics of the `getPriority` and `setPriority` methods must be modified.

4.1.1.1 Setting Priority

The `setPriority` method has the following additional requirements.

- Use of `Thread.setPriority()` shall not affect the correctness of the priority inversion avoidance algorithms controlled by `PriorityCeilingEmulation` and `PriorityInheritance`. Changes to the base priority of a realtime thread as a result of invoking `Thread.setPriority()` are governed by semantics from Chapter 7 on *Synchronization*.
- realtime threads may use `setPriority` to apply the expanded range of priorities available to realtime threads. If a realtime thread's priority parameters object is not shared, `setPriority` behaves effectively as if it included the following code snippet:

```
PriorityParameters pp = getSchedulingParameters();
pp.setPriority(newPriority);
```

- If the realtime thread's priority parameters object is shared with other schedulables, `setPriority` must give the thread an unshared `PriorityParameters` instance allocated in the same memory area as the realtime thread object and containing the new priority value.

- `setPriority` throws `IllegalArgumentException` if the thread is a realtime thread and the new priority is outside the range allowed by the realtime thread's scheduler.
- `setPriority` throws `ClassCastException` if the thread is a realtime thread and its current scheduling parameters object is not an instance of `PriorityParameters`.

4.1.1.2 Getting Priority

The `getPriority` method has the following additional requirements.

- When used on a realtime thread, `getPriority` behaves effectively as if it included the following code snippet:

```
(PriorityParameters)t.getSchedulingParameters()).getPriority()};
```

- If the scheduling parameters are not of type `PriorityParameters`, then a `ClassCastException` is thrown.

All supported monitor control policies must apply to Java threads as well as to all schedulables.

4.1.2 Thread Groups

Thread groups are rooted at a base `ThreadGroup` object which may be created in heap or immortal memory. All thread group objects hold references to all their member threads, and subgroups, and a reference to their parent group. Since heap and immortal memory can not hold references to scoped memory, it follows that a thread group can never be allocated in scoped memory. It then follows that no thread allocated in scoped memory may be referenced from any thread group, and consequently such threads are not part of any thread group and will hold a null thread group reference. Similarly, a no-heap`RealtimeThread` can not be a member of a heap allocated thread group.

1. Realtime threads with `null` thread groups are not included in any operation on any thread group. This applies to enumeration and interruption, as well as the deprecated actions `stop`, `resume`, and `suspend`. However, when the current thread is a realtime thread with a `null` thread group:
   - The `Thread.enumerate` class method returns the integer 1, and populates its array argument with the current realtime thread.
   - `Thread.activeCount` returns 1.

- `Thread.getThreadGroup` returns null in all cases, not only when the thread has terminated.

2. A Java thread (not a realtime thread) that is created from a realtime thread without an explicit thread group and is not assigned a thread group by the security manager inherits the thread group of the realtime thread, if it has one; otherwise an attempt is made to add it to the application root thread group. The constructor shall throw a `SecurityException` if the Java thread is not permitted to use the application root thread group.
3. The thread group of a Java thread created by an asynchronous event handler is assigned as if it was created by a realtime thread without a thread group (as described in 2. above)
4. A thread group cannot be created in scoped memory. The constructor shall throw an `IllegalAssignmentError`.
5. Setting a maximum priority on a thread group, either explicitly or via it parent, has no influence on the realtime threads in that group.
6. Except as specified previously, realtime threads have the same `ThreadGroup` membership rules as the parent `Thread` class.

4.1.3 Current Thread

In Java, the currently executing thread can always be determined by calling the static method `Thread.currentThread()`. In the `RTSJ`, there are two types of schedulable entities: threads and asynchronous event handlers. The latter may be mapped dynamically by the realtime Java virtual machine onto the underlying thread model. The method `Thread.currentThread()` when called from an unbound asynchronous event handler will return the thread that is being used as the current execution engine for that event handler. The program should not rely on this being constant for the lifetime of the program. It can rely on it being constant for the current *release* of the handler (see 6.2 for the definition of a *release*). It is not recommended that the program perform any operations on this underlying thread as it may have an impact beyond that of the current event handler. This also means that thread local memory cannot be relied on in unbound event handlers, because data saved in one release may not be available in the next release.

4.2 Java Memory Model

Some aspects of the Java Memory Model must be tightened for this specification, in particular with regards to interactions with native code or when using the Device Module. A conforming implementation must ensure that volatile loads and stores, raw memory operations (see 12.3.1), and RawBufferFactory fence methods are all ordered in a way that is consistent with respect to native code or hardware devices using platform-native memory coherence protocols to access raw memory or raw byte buffers shared with the virtual machine.

4.3 InterruptedException

The specification extends the use of the `InterruptedException` to support asynchronous transfer of control.

The interruptible methods in the standard libraries (such as Object.wait, Thread.sleep, and Thread.join) have their contract expanded slightly such that they will respond to interruption not only when the interrupt method is invoked on the current thread, but also, for schedulables, when executing within a call to `AIE.doInterruptible` and that AIE is fired. See Chapter 8 on Asynchrony.

4.4 Memory Management

The specification provides for two means of managing memory: garbage collection and special memory areas, which are not collected by the garbage collector. Since memory allocated in Java is always in the heap, or at least appears to be, the initial allocation area is the heap. Furthermore, the allocation area can only be changed either by entering another memory area or by calling a method that explicitly causes allocation in another area. When this alternate memory management module is not present, the conventional java semantics for allocation prevails.

4.4.1 Memory Areas

Using a conventional class in a memory area other than a heap can result in unexpected behavior. This is particularly the case when a method of a class is called in a different area than the object was created, which can lead to exceptions. In general, memory areas other than the heap may become full much faster than expected, because objects that are no longer referenced will not be collected automatically.

A method that allocates an object or takes an object that was created in a different memory area and tries to assign it to a field of its associated object can fail. For example, creating a `List` on the heap and adding to it from a scoped memory will most likely cause an exception. Although using other memory areas such as scoped memory is useful for helping to improving determinism, its use complicates the logic of application and library code.

On systems that support memory areas other than heap and do not support realtime garbage collection, some global resources must be put in immortal memory.

System properties and their `String` values allocated during system initialization shall be allocated in immortal memory. For such a system, class objects should also be stored there. Though this avoids priority inversion with the garbage collector, it can cause higher memory use than expected.

4.4.2 Garbage Collection

Garbage collection is an important safety feature of the Java language and runtime environment. Unfortunately, the garbage collection process can interfere with a realtime program's ability to always meet its timing deadlines. This specification provides two main means of circumventing this problem: using a realtime garbage collector or using the memory area module as an alternative to garbage collection for realtime code. Additionally, an implementation may ignore the problem for an implementation meant as a development system or for systems that choose not to provide realtime guarantees. In any case, an implementation must document what realtime guarantees it gives and which method it uses to do so.

4.4.3 Realtime Garbage Collections

Industrial realtime garbage collectors are available with varying approaches to providing realtime response. Though new collectors will undoubtably be developed, all current ones use a variant of the mark-and-sweep algorithm. In all cases, the collectors are incremental: realtime response is obtained by limiting how much of a collection cycle is done each time the collector runs.

4.4.3.1 Thread-Based Collectors

A realtime thread-based collector is an incremental garbage collector that has its own thread of control and runs at intervals. In this case, the garbage collector needs to be scheduled to ensure that it runs often enough and long enough at each interval to recycle discarded objects fast enough to keep up with allocations. There should also be some maximum time after which the garbage collector can be interrupted.

4.4.3.2 Allocation-Based Collectors

A realtime allocation-based garbage collector does not have its own thread of control. Instead, some interval of garbage collection work is done at each allocation. This work is generally a function of the size of the object being allocated. This work becomes part of the execution time of the program. Again, there should be some maximum time after which the garbage collector can be interrupted.

4.4.3.3 Alternatives to Garbage Collection

This specification provide an Alternative Memory Areas Module for managing memory without garbage collection. An implementation of this specification may provide realtime response by requiring applications to use that module instead of providing a realtime garbage collector. This means that all realtime threads would have to run above the priority of the garbage collector and all communication with conventional threads would have to use some nonblocking protocol.

4.4.3.4 Developer Implementation

An implementation that simply provides all the API but no realtime guarantee is also permitted. This is useful as a development environment. Also, many of the APIs are useful event in a convention Java implementation.

4.5 Summary

The refinements and changes to the semantics of the Java runtime environment and classes shall not affect the functional correctness of Java code written for a conventional Java implementation when running on a Java runtime environment which implements this specification. There may be changes in the relative timing of threads, but these should not violate the conventional Java specifications. The use of some `RTSJ` features with code written for a conventional Java implementation may, however, be surprising. This is particularly true when using alternate memory areas, asynchronous transfer of control, and thread local memory in conjunction with unbound asynchronous event handlers.

Chapter 5

Realtime Threads

This section describes the realtime thread classes. This classes provides for the creation of

- realtime threads that have more precise scheduling semantics than `java.lang.Thread`, and
- realtime threads that have no dependency on the heap.

The `RealtimeThread` class extends `java.lang.Thread`. The `ReleaseParameters, SchedulingParameters`, `MemoryParameters` and `ProcessingGroupParameters` objects passed to the `RealtimeThread` constructor allow the temporal and processor demands of the thread to be communicated to the *scheduler*. The `ConfigurationParameters` class defines, amongst other things, the size of Java thread stack; it also indicates whether the `RealtimeThread` is allowed to allocate or even reference objects from the Java heap (and can thus safely execute in preference to the garbage collector). The `PhasingPolicy` class defines the relationship between the threads start time and its first release time when the start time is in the past.

5.1 Overview

The `RTSJ` provides two types of objects which implement the `Schedulable` interface: realtime threads and asynchronous event handlers. This chapter defines the facilities that are available to realtime threads. In many cases these facilities are also available to asynchronous event handlers. In particular:

- the default scheduler must support the scheduling of both realtime threads and asynchronous event handlers;
- realtime threads and asynchronous event handlers are allowed to enter into memory areas and consequently they have associated scope stacks;
- the flow of control of realtime threads and asynchronous event handlers are affected by the `RTSJ` asynchronous transfer of control facilities;

Where the semantics apply to both realtime threads and asynchronous event handlers, the term *schedulable* will be used.

5.2 Semantics

1. Garbage collection executing in the context of a Java thread must not in itself block execution of a no-heap thread with a higher execution eligibility; however application locks work as specified even when the lock causes synchronization between a heap-using thread and a no-heap thread.
2. Each realtime thread has an attribute which indicates whether an `AsynchronouslyInterruptedException` is pending. This attribute is set when a call to `RealtimeThread.interrupt()` is made on the associated realtime thread, and when an asynchronously interrupted exception's `fire` method is invoked between the time the realtime thread has entered that exception's `doInterruptible` method, and before it has return from `doInterruptible`. (See Chapter 8 on *Asynchrony*.)
3. A call to `RealtimeThread.interrupt()` generates the system's generic `AsynchronouslyInterruptedException`. (See Chapter 8 on *Asynchrony*.)
4. The `RealtimeThread.waitForNextPeriod`, `RealtimeThread.waitForNextRelease`, `RealtimeThread.waitForNextPeriodInterruptible` and `RealtimeThread.waitForNextReleaseInterruptible` methods are for use by realtime threads that have periodic or aperiodic release parameters. In the absence of any deadline miss or cost overrun, or an interrupt in the case of `waitForNextPeriodInterruptible` and `waitForNextReleaseInterruptible`, the methods return when the realtime thread's next period/release is due.
5. In the presence of a cost overrun or a deadline miss, the behavior of `waitForNextPeriod` and `waitForNextRelease`, and their interruptible versions, is governed by the thread's scheduler.
6. The first release time of a realtime thread is governed by: the value of any `start` time in its associated `ReleaseParameter` object and the time at which the `RealtimeThread.start` method is called (or the `RealtimeThread.startPeriodic` method is called and the value of any `PhasingPolicy` parameter passed to it).
7. Instances of `RealtimeThread` that are created in scoped memory and instances of `RealtimeThread` with the the no-heap flag set do not have conventional references to thread groups nor do thread groups have conventional references to these threads. For the purposes of this version of the specification, those references are `null`.
8. Realtime threads with null thread groups respond to uncaught exceptions as follows:

- when the exception is a subclass of `ThreadDeath`, the thread simply terminates,
- otherwise the thread prints a stack trace of the exception to `System.err` before it terminates.

9. System-related termination activity (such as execution of finalizers for scoped objects in scoped memory areas that become unreferenced) triggered by termination of a realtime thread is not subject to cost enforcement or deadline miss detection.

5.3 Package javax.realtime

5.3.1 Enumerations

5.3.1.1 PhasingPolicy

This class defines a set of constants that specify the supported policies for starting a thread or periodic timer, when it is started later than the assigned absolute time. The following table specifies the effective start time (that is, the first release time of a periodic real-time thread). The effective start time of a periodic timer is similar; where the first firing is equivalent to the first release.
Available since RTSJ 2.0

Inheritance

java.lang.Object
  java.lang.Enum
    javax.realtime.PhasingPolicy

5.3.1.1.1 Enumeration Constants

STRICT_PHASING

```
public static final STRICT_PHASING
```

ADJUST_FORWARD

```
public static final ADJUST_FORWARD
```

ADJUST_BACKWARD

```
public static final ADJUST_BACKWARD
```

ADJUST_TO_START

```
public static final ADJUST_TO_START
```

Table 5.1: Effect of PhasingPolicy on the First Release of a RealtimeThread with PeriodicParameters

	STRICT PHASING	**ADJUST FORWARD**	**ADJUST BACKWARD**	**ADJUST TO START**
Relative Time	The time of start method invocation plus `start` time.	The time of start method invocation plus `start` time.	The time of start method invocation plus `start` time.	The time of start method invocation plus `start` time.
Absolute Time, earlier than call to `start`	The `start` method throws an exception.	All releases before the time `start` is called are ignored. The first release is at the start time plus the smallest multiple of `period` whose time is after the time `start` was called.	The first release occurs immediately and the next release is at the start time plus the smallest multiple of `period` whose time is after the time `start` was called.	Release immediately and set next release time to be at the time the start method was invoked plus `period`.
Absolute Time, later than call to `start`	First release is at time passed to `start`.	First release is at time passed to `start`.	First release is at time passed to `start`.	First release is at time passed to `start`.
Without Time	First release is at time of start method invocation	First release is at time of start method invocation	First release is at time of start method invocation	First release is at time of start method invocation

5.3.1.1.2 Methods

values

Signature

```
public static
javax.realtime.PhasingPolicy[] values()
```

valueOf(String)

Signature

```
public static
javax.realtime.PhasingPolicy valueOf(String name)
```

5.3.2 Classes

5.3.2.1 ConfigurationParameters

Configuration parameters provide a way to specify various implementation-dependent parameters such as the Java stack and native stack sizes, and to configure the statically allocated `ThrowBoundaryError`[1] associated with a `Schedulable`[2].

Note that these parameters are immutable.

Available since RTSJ 2.0

Inheritance

java.lang.Object
  javax.realtime.ConfigurationParameters

5.3.2.1.1 Constructors

[1] Section 14.2.3.6
[2] Section 6.4.1.2

ConfigurationParameters(MemoryArea, boolean, int, int, long[])

Creates a parameter object for initializing the state of a `Schedulable`[3]. The parameters provide the data for this initialization. For `RealtimeThread`[4] and bound versions of `AbstractAsyncEventHandler`[5], the stack and message buffers can be set exactly, but for the unbound event handlers, the system cannot give any guarentees to allow thread sharing.

Signature

```
public
  ConfigurationParameters(MemoryArea area, boolean mayUseHeap, int messageLengt

 throws IllegalStateException
```

Parameters

area Specifies the `MemoryArea`[6] in which a thread should start or a handler should be executed. When `null`, the memory area will be that of the current `Thread` or `Schedulable`.

mayUseHeap indicates whether or not the schedulable may use the heap.

messageLength Memory space in bytes dedicated to the message associated with `Schedulable`[7] objects created with these parameters' preallocated exceptions, plus references to the method names/identifiers in the stack trace. The value `0` indicates that no message should be stored. The value of `-1` uses the system default.

stackTraceLength Length of the stack trace buffer dedicated to `Schedulable`[8] objects created with these parameters' preallocated exceptions, in frames. The amount of space this requires is implementation-specific. The value `0` indicates that no stack trace should be stored. The value of `-1` uses the system default.

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

Throws

[3] Section 6.4.1.2
[4] Section 5.3.2.2
[5] Section 8.4.3.2
[6] Section 11.4.3.6
[7] Section 6.4.1.2
[8] Section 6.4.1.2

IllegalStateException when `mayUseHead` is `false` and `area` is an instance of `HeapMemory`[9].

ConfigurationParameters(MemoryArea, boolean, int, int)

Same as `ConfigurationParameters(MemoryArea,boolean,int,int,long[])`[10] with arguments `area, mayUseHeap, messageLength, stackTraceLength, null`.

Signature

```
public
  ConfigurationParameters(MemoryArea area, boolean mayUseHeap, int messageLength, in
```

ConfigurationParameters(MemoryArea, boolean)

Same as `ConfigurationParameters(MemoryArea,boolean,int,int,long[])`[11] with arguments `area, true, -1, -1, null`.

Signature

```
public
  ConfigurationParameters(MemoryArea area, boolean mayUseHeap)
```

ConfigurationParameters(MemoryArea)

Same as `ConfigurationParameters(MemoryArea,boolean,int,int,long[])`[12] with arguments `area, true, -1, -1, null`.

Signature

```
public
  ConfigurationParameters(MemoryArea area)
```

[9] Section 11.4.3.1
[10] Section 5.3.2.1.1
[11] Section 5.3.2.1.1
[12] Section 5.3.2.1.1

ConfigurationParameters(long[])

Same as `ConfigurationParameters(MemoryArea,boolean,int,int,long[])`[13] with arguments `HeapMemory.instance(), true, -1, -1, sizes`.

Signature

```
public
  ConfigurationParameters(long[] sizes)
```

5.3.2.1.2 Methods

---

getMemoryArea

Gets the initial memory area specified.

Signature

```
public
javax.realtime.MemoryArea getMemoryArea()
```

Returns

the initial memory area, when one is set; otherwise null.

mayUseHeap

Find out if heap access is set.

Signature

```
public
boolean mayUseHeap()
```

Returns

true when the heap may be accessed.

[13] Section 5.3.2.1.1

getMessageLength

Gets the memory space in bytes dedicated to the message associated with `Schedulable`[14] objects created with these parameters' preallocated exceptions, plus references to the method names/identifiers in the stack trace. The value `0` indicates that no message will be stored.

Signature

```
public
int getMessageLength()
```

Returns

Reserved memory size in bytes.

getStackTraceLength

Gets the length of the stack trace buffer dedicated to `Schedulable`[15] objects created with these parameters' preallocated exceptions, in frames. The amount of space this requires is implementation-specific. The value `0` indicates that no stack trace will be stored.

Signature

```
public
int getStackTraceLength()
```

Returns

Reserved memory size in implementation-dependent stack frames.

getSizes

Gets the array of implementation-specific sizes associated with `Schedulable`[16] objects created with these parameters. *This method may allocate memory.*

Signature

```
public
long[] getSizes()
```

[14] Section 6.4.1.2
[15] Section 6.4.1.2
[16] Section 6.4.1.2

Returns

Array of implementation-specific sizes.

5.3.2.2 RealtimeThread

Class `RealtimeThread` extends `Thread` and adds access to realtime services such as asynchronous transfer of control, nonheap memory, and advanced scheduler services.

As with `java.lang.Thread`, there are two ways to create a usable `RealtimeThread`.

- Create a new class that extends `RealtimeThread` and override the `run()` method with the logic for the thread.
- Create an instance of `RealtimeThread` using one of the constructors with a `logic` parameter. Pass a `Runnable` object whose `run()` method implements the logic of the thread.

Inheritance

java.lang.Object
  java.lang.Thread
    javax.realtime.RealtimeThread

Interfaces

BoundSchedulable
AsyncTimable

5.3.2.2.1 Constructors

RealtimeThread(SchedulingParameters, ReleaseParameters, MemoryParameters, ConfigurationParameters, ProcessingGroupParameters, TimeDispatcher, ThreadGroup, Runnable)

Create a realtime thread with the given characteristics and a specified `Runnable`. The thread group of the new thread is inherited from its creator unless the newly-created realtime thread is allocated in scoped memory, then its thread group is (effectively) `null`.

The newly-created realtime thread is associated with the scheduler in effect during execution of the constructor.

The newly-created realtime thread inherits the affinity of its creator unless it was created by a Java thread or an unbound asynchronous event handler. In these cases,

the affinity is that which is returned from `Affinity.getHeapDefault()`[17]. When the newly-created realtime thread has `ProcessingGroupParameters`[18] and the intersection of the `group`'s affinity and the newly-created realtime thread's affinity (as specified above) is null, then the newly-created realtime thread's affinity is set to that which is returned by `Affinity.getProcessingGroupDefault`[19].
Available since RTSJ 2.0

Signature

```
public
  RealtimeThread(SchedulingParameters scheduling, ReleaseParameters release, MemoryP
```

Parameters

scheduling The `SchedulingParameters`[20] associated with `this` (And possibly other instances of `Schedulable`[21]). When `scheduling` is `null` and the creator is a schedulable, `SchedulingParameters`[22] is a clone of the creator's value created in the same memory area as `this`. When `scheduling` is `null` and the creator is a Java thread, the contents and type of the new `SchedulingParameters` object is governed by the associated scheduler.

release The `ReleaseParameters`[23] associated with `this` (and possibly other instances of `Schedulable`[24]). When `release` is `null` the new `RealtimeThread` will use a clone of the default `ReleaseParameters` for the associated scheduler created in the memory area that contains the `RealtimeThread` object.

memory The `MemoryParameters`[25] associated with `this` (and possibly other instances of `Schedulable`[26]). When `memory` is `null`, the new `RealtimeThread` receives `null` value for its memory parameters, and the amount or rate of memory allocation for the new thread is unrestricted.

pgp The `ProcessingGroupParameters`[27] associated with this (and possibly

[17]Section 6.4.2.1.2
[18]Section 6.4.2.7
[19]Section 6.4.2.1.2
[20]Section 6.4.2.10
[21]Section 6.4.1.2
[22]Section 6.4.2.10
[23]Section 6.4.2.8
[24]Section 6.4.1.2
[25]Section 11.4.3.7
[26]Section 6.4.1.2
[27]Section 6.4.2.7

other instances of `Schedulable`[28]). When `null`, the new `RealtimeThread` will not be associated with any processing group.

sizing The `ConfigurationParameters`[29] associated with `this` (and possibly other instances of `Schedulable`[30]. When `sizing` is `null`, this `RealtimeThread` will reserve no space for preallocated exceptions and implementation-specific values will be set to their implementation-defined defaults.

dispatcher The `TimeDispatcher`[31] to use for realtime sleep and determining the period of a periodic thread.

group The `ThreadGroup` of the newly created realtime thread.

logic The `Runnable` object whose `run()` method will serve as the logic for the new `RealtimeThread`. When `logic` is `null`, the `run()` method in the new object will serve as its logic.

Throws

IllegalArgumentException when the parameters are not compatible with the associated scheduler.

IllegalAssignmentError when the new `RealtimeThread` instance cannot hold a reference to any of the values of `scheduling release memory` or `group`, when those parameters cannot hold a reference to the new `RealtimeThread`, when the new `RealtimeThread` instance cannot hold a reference to the values of `area` or `logic`, when the initial memory area is not specified and the new `RealtimeThread` instance cannot hold a reference to the default initial memory area, and when the thread may not use the heap, as specified by its configuration, and any of the following is true:

- the initial memory are is not specified,
- the initial memory is heap memory,
- the initial memory area, scheduling release, memory, or group is allocated in heap memory.
- when this is in heap memory, or
- logic is in heap memory.

RealtimeThread(SchedulingParameters, ReleaseParameters, MemoryParameters, ConfigurationParameters, ProcessingGroupParameters, Runnable)

Create a realtime thread with the given `SchedulingParameters`[32], `ReleasePar-`

[28]Section 6.4.1.2
[29]Section 5.3.2.1
[30]Section 6.4.1.2
[31]Section 10.4.2.4
[32]Section 6.4.2.10

ameters[33], MemoryParameters[34], MemoryArea[35] ProcessingGroupParameters[36], ConfigurationParameters[37], a specified `Runnable`, and default values for all other parameters.

This constructor is equivalent to `RealtimeThread(scheduling, release, memory, configuration, group, null, null, logic)`.

Available since RTSJ 2.0 Create a realtime thread with the given characteristics and a specified `Runnable`. The thread group of the new thread is inherited from its creator unless the newly-created realtime thread is allocated in scoped memory, then its thread group is (effectively) `null`.

Signature

```
public
  RealtimeThread(SchedulingParameters scheduling, ReleaseParameters release, MemoryP
```

RealtimeThread(SchedulingParameters, ReleaseParameters, ConfigurationParameters, Runnable)

Create a realtime thread with the given SchedulingParameters[38], ReleaseParameters[39], MemoryArea[40] and a specified `Runnable` and default values for all other parameters.

This constructor is equivalent to `RealtimeThread(scheduling, release, configuration, null, null, null, logic)`.

Available since RTSJ 2.0

Signature

```
public
  RealtimeThread(SchedulingParameters scheduling, ReleaseParameters release, Configu
```

[33]Section 6.4.2.8
[34]Section 11.4.3.7
[35]Section 11.4.3.6
[36]Section 6.4.2.7
[37]Section 5.3.2.1
[38]Section 6.4.2.10
[39]Section 6.4.2.8
[40]Section 11.4.3.6

RealtimeThread(SchedulingParameters, ReleaseParameters, ConfigurationParameters)

Create a realtime thread with the given SchedulingParameters[41], ReleaseParameters[42] and MemoryArea[43] and default values for all other parameters.

This constructor is equivalent to `RealtimeThread(scheduling, release, null, area, null, null, null, null)`.

Available since RTSJ 2.0

Signature

```
public
  RealtimeThread(SchedulingParameters scheduling, ReleaseParameters release, Co
```

RealtimeThread(SchedulingParameters, ReleaseParameters, Runnable)

Create a realtime thread with the given SchedulingParameters[44], ReleaseParameters[45] and a specified `Runnable` and default values for all other parameters.

This constructor is equivalent to `RealtimeThread(scheduling, release, null, null, null, nll, null, logic)`.

Available since RTSJ 2.0

Signature

```
public
  RealtimeThread(SchedulingParameters scheduling, ReleaseParameters release, Ru
```

[41] Section 6.4.2.10
[42] Section 6.4.2.8
[43] Section 11.4.3.6
[44] Section 6.4.2.10
[45] Section 6.4.2.8

RealtimeThread(SchedulingParameters, ReleaseParameters, MemoryParameters, MemoryArea, ProcessingGroupParameters, Runnable)

Create a realtime thread with the given characteristics and a Runnable. This is equivalent to `RealtimeThread(scheduling, release, memory, new ConfigurationParameters(area), group, null, null, logic)`.

Signature

```
public
  RealtimeThread(SchedulingParameters scheduling, ReleaseParameters release, MemoryP
```

RealtimeThread(SchedulingParameters, ReleaseParameters)

Create a realtime thread with the given `SchedulingParameters`[46] and `ReleaseParameters`[47] and default values for all other parameters.

This constructor is equivalent to `RealtimeThread(scheduling, release, null, null, null, null, null)`.

Signature

```
public
  RealtimeThread(SchedulingParameters scheduling, ReleaseParameters release)
```

RealtimeThread(SchedulingParameters, TimeDispatcher)

Create a realtime thread with the given `SchedulingParameters`[48] and `TimeDispatcher`[49] and default values for all other parameters. This constructor is equivalent to `RealtimeThread(scheduling, null, null, null, null, dispatcher, null, null)`.

[46]Section 6.4.2.10
[47]Section 6.4.2.8
[48]Section 6.4.2.10
[49]Section 10.4.2.4

Available since RTSJ 2.0

Signature

```
public
  RealtimeThread(SchedulingParameters scheduling, TimeDispatcher dispatcher)
```

RealtimeThread(SchedulingParameters)

Create a realtime thread with the given `SchedulingParameters`[50] and default values for all other parameters. This constructor is equivalent to `RealtimeThread(scheduling, null, null, null, null, null, null, null)`.

Signature

```
public
  RealtimeThread(SchedulingParameters scheduling)
```

RealtimeThread

Create a realtime thread with default values for all parameters. This constructor is equivalent to `RealtimeThread(null, null, null, null, null, null, null)`.

Signature

```
public
  RealtimeThread()
```

5.3.2.2.2 Methods

[50]Section 6.4.2.10

currentRealtimeThread

Gets a reference to the current instance of `RealtimeThread`.

It is permissible to call `currentRealtimeThread` when control is in an `AsyncEventHandler`[51]. The method will return a reference to the `RealtimeThread` supporting that release of the async event handler.

Signature

```
public static
javax.realtime.RealtimeThread currentRealtimeThread()
```

Throws

ClassCastException when the current execution context is that of a Java thread.

Returns

A reference to the current instance of `RealtimeThread`.

getCurrentMemoryArea

Return a reference to the `MemoryArea`[52] object representing the current allocation context.

When this method is invoked from a Java thread it will return that thread's current memory area (heap or immortal memory).

Signature

```
public static
javax.realtime.MemoryArea getCurrentMemoryArea()
```

Returns

A reference to the `MemoryArea`[53] object representing the current allocation context.

getInitialMemoryAreaIndex

Gets the position of the initial memory area for the current `Schedulable`[54] in the memory area stack. Memory area stacks may include inherited stacks from

[51] Section 8.4.3.5
[52] Section 11.4.3.6
[53] Section 11.4.3.6
[54] Section 6.4.1.2

parent threads. The initial memory area of a `RealtimeThread`[55] or an `AbstractAsyncEventHandler`[56] is the memory area specified in its constructor. The index of the initial memory area in the initial memory area stack is a fixed property of a `Schedulable`.

When the current memory area stack of the current realtime thread is not the original stack and the memory area at the initial memory area index is not the initial memory area, then `IllegalStateException` is thrown.

Signature

```
public static
int getInitialMemoryAreaIndex()
```

Throws

IllegalStateException when the memory area stack of the current `RealtimeThread` has changed from its initial configuration and the memory area at the originally specified initial memory area index is not the initial memory area.

ClassCastException when the current execution context is that of a Java thread.

Returns

The index into the initial memory area stack of the initial memory area of the current `RealtimeThread`.

getMemoryAreaStackDepth

Gets the size of the stack of `MemoryArea`[57] instances to which the current schedulable has access.

Note: The current memory area (`getCurrentMemoryArea()`[58]) is found at memory area stack index of `getMemoryAreaStackDepth() - 1`.

Signature

```
public static
int getMemoryAreaStackDepth()
```

Throws

ClassCastException when the current execution context is that of a Java thread.

Returns

The size of the stack of `MemoryArea`[59] instances.

[55] Section 5.3.2.2
[56] Section 8.4.3.2
[57] Section 11.4.3.6
[58] Section 5.3.2.2.2
[59] Section 11.4.3.6

getOuterMemoryArea(int)

Gets the instance of `MemoryArea`[60] in the memory area stack at the index given. When the given index does not exist in the memory area scope stack then `null` is returned.

Note: The current memory area (`getCurrentMemoryArea()`[61]) is found at memory area stack index `getMemoryAreaStackDepth() - 1`, so `getCurrentMemoryArea() == getOutMemoryArea(getMemoryAreaStackDepth() - 1)`.

Signature

```
public static
javax.realtime.MemoryArea getOuterMemoryArea(int index)
```

Parameters

index The offset into the memory area stack.

Throws

ClassCastException when the current execution context is that of a Java thread.

MemoryAccessError when the memory area is allocate in heap memory and the caller is a schedulable that may not use the heap.

Returns

The instance of `MemoryArea`[62] at index or `null` when the given value does not correspond to a position in the stack.

sleep(javax.realtime.HighResolutionTime<T>)

A sleep method that is controlled by the realtime clock.

Equivalent to `sleep(Clock.getRealtimeClock(), time)`

Signature

```
public static
void sleep(javax.realtime.HighResolutionTime<T> time)
throws InterruptedException
```

waitForNextPeriod

[60] Section 11.4.3.6
[61] Section 5.3.2.2.2
[62] Section 11.4.3.6

Causes the current realtime thread to delay until the beginning of the next period. Used by threads that have a reference to a ReleaseParameters[63] type of PeriodicParameters[64] to block until the start of each period. The first period starts when this thread is first released. Each time it is called this method will block until the start of the next period unless the thread is in a deadline miss condition. In that case the operation of waitForNextPeriod is controlled by this thread's scheduler. (See PriorityScheduler[65].)

Available since RTSJ 1.0.1 Changed from an instance method to a static method.

Signature

```
public static
boolean waitForNextPeriod()
```

Throws

IllegalThreadStateException when this does not have a reference to a ReleaseParameters[66] type of PeriodicParameters[67].

ClassCastException when the current thread is not an instance of RealtimeThread.

Returns

True when the thread is not in a deadline miss condition. Otherwise the return value is governed by this thread's scheduler.

waitForNextPeriodInterruptible

The waitForNextPeriodInterruptible() method is a duplicate of waitForNextPeriod()[68] except that waitForNextPeriodInterruptible is able to throw InterruptedException.

Used by threads that have a reference to a ReleaseParameters[69] type of PeriodicParameters[70] to block until the start of each period. The first period starts when this thread is first released. Each time it is called this method will block until the start of the next period unless the thread is in a deadline miss condition.

[63] Section 6.4.2.8
[64] Section 6.4.2.4
[65] Section 6.4.2.6
[66] Section 6.4.2.8
[67] Section 6.4.2.4
[68] Section 5.3.2.2.2
[69] Section 6.4.2.8
[70] Section 6.4.2.4

In that case the operation of `waitForNextPeriodInterruptible` is controlled by this thread's scheduler. (See `PriorityScheduler`[71].)
Available since RTSJ 1.0.1

Signature

```
public static
boolean waitForNextPeriodInterruptible()
throws InterruptedException
```

Throws

InterruptedException when the thread is interrupted by `interrupt()`[72] or `AsynchronouslyInterruptedException.fire()`[73] during the time between calling this method and returning from it.
An interrupt during `waitForNextPeriodInterruptible` is treated as a release for purposes of scheduling. This is likely to disrupt proper operation of the periodic thread. The periodic behavior of the thread is unspecified until the state is reset by altering the thread's periodic parameters.
ClassCastException when the current thread is not an instance of `RealtimeThread`.
IllegalThreadStateException when `this` does not have a reference to a `ReleaseParameters`[74] type of `PeriodicParameters`[75].

Returns

True when the thread is not in a deadline miss condition. Otherwise the return value is governed by this thread's scheduler.

waitForNextRelease

Causes the current realtime thread to delay until the next release. (See `release()`[76].) Used by threads that have a reference to aperiodic `ReleaseParameters`[77] The first release starts when `this` thread is released as a consequence of the action of one of the `start()`[78] family of methods. Each time it is called this method will block until the next release unless the thread is in a deadline miss condition.

[71] Section 6.4.2.6
[72] Section 5.3.2.2.2
[73] Section 8.4.2.1.2
[74] Section 6.4.2.8
[75] Section 6.4.2.4
[76] Section 5.3.2.2.2
[77] Section 6.4.2.8
[78] Section 5.3.2.2.2

In that case the operation of `waitForNextRelease` is controlled by this thread's scheduler. (See `PriorityScheduler`[79].)
Available since RTSJ 2.0

Signature

`public static`
`boolean waitForNextRelease()`

Throws

IllegalThreadStateException when `this` does not have a reference to a `ReleaseParameters`[80] type of `AperiodicParameters`[81].

ClassCastException when the current thread is not an instance of `RealtimeThread`.

Returns

True when the thread is not in a deadline miss condition. Otherwise the return value is governed by this thread's scheduler.

waitForNextReleaseInterruptible

Causes the current realtime thread to delay until the next release. (See `release()`[82].) Used by threads that have a reference to aperiodic `ReleaseParameters`[83]. The first release starts when `this` thread is released as a consequence of the action of one of the `start()`[84] family of methods. Each time it is called this method will block until the next release unless the thread is in a deadline miss condition. In that case the operation of `waitForNextRelease` is controlled by this thread's scheduler. (See `PriorityScheduler`[85].)
Available since RTSJ 2.0

Signature

`public static`
`boolean waitForNextReleaseInterruptible()`

[79] Section 6.4.2.6
[80] Section 6.4.2.8
[81] Section 6.4.2.2
[82] Section 5.3.2.2.2
[83] Section 6.4.2.8
[84] Section 5.3.2.2.2
[85] Section 6.4.2.6

```
throws InterruptedException
```

Throws

InterruptedException

IllegalThreadStateException when `this` does not have a reference to a `ReleaseParameters`[86] type of `AperiodicParameters`[87].

ClassCastException when the current thread is not an instance of `RealtimeThread`.

Returns

True when the thread is not in a deadline miss condition. Otherwise the return value is governed by this thread's scheduler.

release

Generate a release for this `RealtimeThread`. The action of this release is governed by the schedule. It may, for instance, act immediately, or be queued, delayed, or discarded.

Available since RTSJ 2.0

Signature

```
public
void release()
```

Throws

IllegalThreadStateException when `this` does not have a reference to a `ReleaseParameters`[88] type of `AperiodicParameters`[89].

deschedule

When the `ReleaseParameters`[90] object associated with `this RealtimeThread` is an instance of is `AperiodicParameters`[91], perform any *deschedule* actions specified by this thread's scheduler. When the type of the associated instance of `ReleaseParameters`[92] is not `AperiodicParameters`[93] nothing happens.

[86] Section 6.4.2.8
[87] Section 6.4.2.2
[88] Section 6.4.2.8
[89] Section 6.4.2.2
[90] Section 6.4.2.8
[91] Section 6.4.2.2
[92] Section 6.4.2.8
[93] Section 6.4.2.2

Available since RTSJ 2.0

Signature

```
public
void deschedule()
```

deschedulePeriodic

When the `ReleaseParameters`[94] object associated with `this RealtimeThread` is an instance of `PeriodicParameters`[95], perform any *deschedulePeriodic* actions specified by this thread's scheduler. When the type of the associated instance of `ReleaseParameters`[96] is not `PeriodicParameters`[97] nothing happens.

Signature

```
public
void deschedulePeriodic()
```

getMemoryArea

Return the initial memory area for this `RealtimeThread`. When not specified through the constructor, the default is a *reference* to the current allocation context when `this` was constructed.
Available since RTSJ 1.0.1

Signature

```
public
javax.realtime.MemoryArea getMemoryArea()
```

Returns

A reference to the initial memory area for this thread.

[94] Section 6.4.2.8
[95] Section 6.4.2.4
[96] Section 6.4.2.8
[97] Section 6.4.2.4

mayUseHeap

Test if the thread may access the heap.
Available since RTSJ 2.0

Signature

```
public
boolean mayUseHeap()
```

getMemoryParameters

@inheritDoc

Signature

```
public
javax.realtime.MemoryParameters getMemoryParameters()
```

Returns

@inheritDoc

getProcessingGroupParameters

@inheritDoc

Signature

```
public
javax.realtime.ProcessingGroupParameters
getProcessingGroupParameters()
```

Returns

@inheritDoc

getConfigurationParameters

@inheritDoc

Signature

```
public
javax.realtime.ConfigurationParameters
getConfigurationParameters()
```

Returns

@inheritDoc

getReleaseParameters

@inheritDoc

Signature

```
public
javax.realtime.ReleaseParameters getReleaseParameters()
```

Returns

@inheritDoc

getScheduler

@inheritDoc

Signature

```
public
javax.realtime.Scheduler getScheduler()
```

Returns

@inheritDoc

getSchedulingParameters

@inheritDoc

Signature

```
public
javax.realtime.SchedulingParameters getSchedulingParameters()
```

Returns

@inheritDoc

interrupt

Extends the function of `Thread.interrupt()`, generates the generic AsynchronouslyInterruptedException and targets it at this, and sets the interrupted state to pending. (See `AsynchronouslyInterruptedException`[98].

The semantics of `Thread.interrupt()` are preserved.

Signature

```
public
void interrupt()
```

schedule

Begin unblocking `RealtimeThread.waitForNextRelease`[99] for an periodic thread. When deadline miss detection is disabled, enable it. Typically used when an aperiodic schedulable is in a deadline miss condition.

The details of the interaction of this method with `deschedule`[100], `waitForNextRelease`[101] and `release`[102] are dictated by this thread's scheduler. When this `RealtimeThread` does not have a type of `AperiodicParameters`[103] as its `ReleaseParameters`[104] nothing happens.

Available since RTSJ 2.0

Signature

```
public
void schedule()
```

schedulePeriodic

Begin unblocking `RealtimeThread.waitForNextPeriod`[105] for a periodic thread. When deadline miss detection is disabled, enable it. Typically used when a periodic

[98] Section 8.4.2.1
[99] Section 5.3.2.2.2
[100] Section 5.3.2.2.2
[101] Section 5.3.2.2.2
[102] Section 5.3.2.2.2
[103] Section 6.4.2.2
[104] Section 6.4.2.8
[105] Section 5.3.2.2.2

schedulable is in a deadline miss condition. The details of the interaction of this method with `deschedulePeriodic`[106] and `waitForNextPeriod`[107] are dictated by this thread's scheduler.

When this `RealtimeThread` does not have a type of `PeriodicParameters`[108] as its `ReleaseParameters`[109] nothing happens.

Signature

```
public
void schedulePeriodic()
```

setMemoryParameters(MemoryParameters)

@inheritDoc

Signature

```
public
void setMemoryParameters(MemoryParameters memory)
```

Parameters

memory @inheritDoc

Throws

IllegalArgumentException @inheritDoc
IllegalAssignmentError @inheritDoc
IllegalThreadStateException @inheritDoc

setProcessingGroupParameters(ProcessingGroupParameters)

@inheritDoc

Signature

```
public
void setProcessingGroupParameters(ProcessingGroupParameters
group)
```

[106] Section 5.3.2.2.2
[107] Section 5.3.2.2.2
[108] Section 6.4.2.4
[109] Section 6.4.2.8

Parameters

group @inheritDoc

Throws

IllegalArgumentException @inheritDoc
IllegalAssignmentError @inheritDoc
IllegalThreadStateException @inheritDoc

setReleaseParameters(ReleaseParameters)

@inheritDoc

Signature

```
public
void setReleaseParameters(ReleaseParameters release)
```

Parameters

release @inheritDoc

Throws

IllegalArgumentException @inheritDoc
IllegalAssignmentError @inheritDoc
IllegalThreadStateException @inheritDoc

setScheduler(Scheduler)

@inheritDoc

Signature

```
public
void setScheduler(Scheduler scheduler)
```

Parameters

scheduler @inheritDoc

Throws

IllegalArgumentException @inheritDoc
IllegalAssignmentError @inheritDoc
SecurityException @inheritDoc
IllegalThreadStateException @inheritDoc

setScheduler(Scheduler, SchedulingParameters, ReleaseParameters, MemoryParameters, ProcessingGroupParameters)

@inheritDoc

Signature

```
public
void setScheduler(Scheduler scheduler, SchedulingParameters
scheduling, ReleaseParameters release, MemoryParameters
memoryParameters, ProcessingGroupParameters group)
```

Parameters

scheduler @inheritDoc
scheduling @inheritDoc
release @inheritDoc
memoryParameters @inheritDoc
group @inheritDoc

Throws

IllegalArgumentException @inheritDoc
IllegalAssignmentError @inheritDoc
IllegalThreadStateException @inheritDoc
SecurityException @inheritDoc

setSchedulingParameters(SchedulingParameters)

@inheritDoc

Signature

```
public
void setSchedulingParameters(SchedulingParameters scheduling)
```

Parameters

scheduling @inheritDoc

Throws

IllegalArgumentException @inheritDoc
IllegalAssignmentError @inheritDoc
IllegalThreadStateException @inheritDoc

startPeriodic(PhasingPolicy)

Start the thread with the specified phasing policy.
Available since RTSJ 2.0

Signature

```
public
void startPeriodic(PhasingPolicy phasingPolicy)
throws LateStartException
```

Parameters

phasingPolicy The phasing policy to be applied when the start time given in the realtime thread's associated `PeriodicParameters`[110] is in the past.

Throws

javax.realtime.LateStartException when the actual start time is after the assigned start time and the phasing policy is `PhasingPolicy.STRICT_PHASING`[111].
IllegalArgumentException when the thread is not periodic, or when its start time is not absolute.

start

Set up the realtime thread's environment and start it. The set up might include delaying it until the assigned start time and initializing the thread's scope stack. (See `ScopedMemory`[112].)

Signature

```
public
void start()
```

getLastReleaseTime

Equivalent to `getLastReleaseTime(null)`
Available since RTSJ 2.0

[110]Section 6.4.2.4
[111]Section 5.3.1.1.1
[112]Section 11.4.3.11

Signature

```
public
javax.realtime.AbsoluteTime getLastReleaseTime()
```

getLastReleaseTime(AbsoluteTime)

Return the absolute time of this thread's last release, whether periodic or aperiodic.

The clock in the returned absolute time shall be the realtime clock for aperiodic releases and the clock used for the periodic release for periodic releases.

Available since RTSJ 2.0

Signature

```
public
javax.realtime.AbsoluteTime getLastReleaseTime(AbsoluteTime
dest)
```

Returns

the last release time in `dest`. When `dest` is `null`, create a new absolute time instance in the current memory area.

getEffectiveStartTime

Equivalent to `getEffectiveStartTime(null)`.

Available since RTSJ 2.0

Signature

```
public
javax.realtime.AbsoluteTime getEffectiveStartTime()
```

getEffectiveStartTime(AbsoluteTime)

Return the effective start time of this realtime thread. This is not necessarily the same as the start time in the release parameters.

- When the release parameters' start time is relative, the effective start time is the time of the first release.

- When the release parameters' start time is an absolute time after start() is invoked, the effective start time is the same as the release parameters' start time.
- When the release parameters' start time is an absolute time before start() is invoked, the effective start time depends on the phasing policy.

The default is to set the effective start time equal to the time start() is invoked.
Available since RTSJ 2.0

Signature

```
public
javax.realtime.AbsoluteTime getEffectiveStartTime(AbsoluteTime
dest)
```

Returns

The effective start time in `dest`. When `dest` is `null`, return the effective start time in an `AbsoluteTime`[113] instance created in the current memory area.

getCurrentConsumption(RelativeTime)

Available since RTSJ 2.0

Signature

```
public static
javax.realtime.RelativeTime getCurrentConsumption(RelativeTime
dest)
```

Throws

IllegalStateException when the caller is not a `RealtimeThread`[114].

Returns

The CPU consumption for this release. When `dest` is `null`, return the CPU consumption in a `RelativeTime`[115] instance created in the current execution context. When `dest` is not `null`, return the CPU consumption in `dest`

getCurrentConsumption

[113] Section 9.4.1.1
[114] Section 5.3.2.2
[115] Section 9.4.1.3

Equivalent to `getCurrentConsumption(null)`.
Available since RTSJ 2.0

Signature

```
public static
javax.realtime.RelativeTime getCurrentConsumption()
```

getMinConsumption(RelativeTime)

Get the minimum CPU consumption measured for any completed release of this thread.
Available since RTSJ 2.0

Signature

```
public
javax.realtime.RelativeTime getMinConsumption(RelativeTime dest)
```

Returns

the minimum CPU consumption in `dest`. When `dest` is `null` return the minimum CPU consumption in a `RelativeTime`[116] instance created in the current memory area.

getMinConsumption

Equivalent to `getMinConsumption(null)`.
Available since RTSJ 2.0

Signature

```
public
javax.realtime.RelativeTime getMinConsumption()
```

getMaxConsumption(RelativeTime)

[116]Section 9.4.1.3

Get the maximum CPU consumption measured for any completed release of this thread.

Available since RTSJ 2.0

Signature

```
public
javax.realtime.RelativeTime getMaxConsumption(RelativeTime dest)
```

Returns

the maximum CPU consumption in `dest`. When `dest` is `null` return the maximum CPU consumption in a `RelativeTime`[117] instance created in the current memory area.

getMaxConsumption

Equivalent to `getMaxConsumption(null)`.

Available since RTSJ 2.0

Signature

```
public
javax.realtime.RelativeTime getMaxConsumption()
```

getDispatcher

Get the dispatcher responsible for handling sleep requests issued by this thread See Section AsyncTimable.getDispatcher()

Available since RTSJ 2.0

Signature

```
public
javax.realtime.TimeDispatcher getDispatcher()
```

[117]Section 9.4.1.3

fire

Used by the {#link Clock} infrastructure to cause a call to `waitForNextPeriod`[118] to return.
See Section AsyncTimable.fire()

Available since RTSJ 2.0

Signature

```
public final
void fire()
```

wakeUp

Used by the {#link Clock} infrastructure to cause a call to `sleep`[119] to return.
See Section Schedulable.wakeUp()

Available since RTSJ 2.0

Signature

```
public final
void wakeUp()
```

5.4 Rationale

The `RTSJ` platform's priority-preemptive dispatching model is very similar to the dispatching model found in the majority of commercial realtime operating systems. However, the dispatching semantics were purposefully relaxed in order to allow execution on a wide variety of operating systems. Thus, it is appropriate to specify realtime threads by extending `java.lang.Thread`.

The `ReleaseParameters` and `MemoryParameters` provided to the `RealtimeThread` constructor allow for a number of common realtime thread types, including periodic threads.

[118] Section 5.3.2.2.2
[119] Section 5.3.2.2.2

The `ConfigurationParameters` class is provided with a no-heap option in order to allow time-critical schedulables to execute in preference to the garbage collector given appropriate assignment of execution eligibility. The memory access and assignment semantics of these no-heap schedulables are designed to guarantee that the execution of such threads does not lead to an inconsistent heap state.

Chapter 6

Scheduling

6.1 Overview

The scheduler required by this specification is fixed-priority preemptive with at least 28 unique priority levels. It is represented by the class `PriorityScheduler` and is called the *base scheduler*.

The schedulables required by this specification are defined by the classes `RealtimeThread`, `AsyncEventHandler`, and `BoundAsyncEventHandler`. The base scheduler assigns processor resources according to the schedulables' release characteristics, execution eligibility, affinity, and processing group values. Subclasses of the schedulables are also schedulables and behave as these required classes.

The scheduler dispatches a schedulable, that is ready to run, on a CPU. Some systems, such as multicore systems, have more than one CPU to choose from. By default, a ready schedulable would be dispatched on the next available CPU; however, the specification provides an interface, `Affinity` to control, on which sets of CPUs a given schedulable may run.

An instance of the `SchedulingParameters` class contains values of execution eligibility. A schedulable is considered to have the execution eligibility represented by the `SchedulingParameters` object currently bound to it. For implementations providing only the base scheduler, the scheduling parameters object is an instance of `PriorityParameters` (a subclass of `SchedulingParameters`).

An instance of the `ReleaseParameters` class or its subclasses, `PeriodicParameters`, `AperiodicParameters`, and `SporadicParameters`, contains values that define a particular release characteristic. A schedulable is considered to have the release characteristics of a single associated instance of the `ReleaseParameters` class.

For a realtime thread, the scheduler defines the behavior of the realtime thread's `waitForNextPeriod`, `waitForNextPeriodInterruptible`, `waitForNextRelease`, and `waitForNextReleaseInterruptible` methods. For all `Schedulables`, the scheduler monitors cost overrun and deadline miss conditions based on its release parameters. Release parameters also govern the treatment of the minimum interarrival time for sporadic schedulables.

An instance of the `ProcessingGroupParameters` class contains values that define a temporal scope for a processing group on a single processor. If a schedulable has an associated instance of the `ProcessingGroupParameters` class, it is said to execute within the temporal scope defined by that instance. A single instance of the `ProcessingGroupParameters` class can be (and typically is) associated with many schedulables. If the implementation supports cost enforcement, the combined processor demand of all of the schedulables associated with an instance of the `ProcessingGroupParameters` class must not exceed the values in that instance (i.e., the defined temporal scope). The processor demand is determined by the `Scheduler`.

The scheduling classes provide the necessary support for realtime scheduling. These classes

- allow the definition of schedulables,
- manage the assignment of execution eligibility to schedulable objects,
- manage the execution of instances of the `AsyncEventHandler` and `RealtimeThread` classes,
- assign release characteristics to schedulables,
- assign execution eligibility values to schedulables, and
- manage the execution of groups of schedulables that collectively exhibit additional release characteristics.

6.2 Definitions

Task is the name given a unit of independent execution. In conventional Java, this is a thread. The `Schedulable` interface marks realtime tasks. The classes that implement `Schedulable` are subject to the scheduling behavior of realtime schedulars. Instances of these classes are referred to as emphSchedulables (SO) and provide three execution states: *executing*, *blocked*, and *eligible-for-execution*.

- *Executing* refers to the state where the schedulable is currently running on a processor.
- *Blocked* refers to the state where the schedulable is not among those schedulables that could be selected to have their state changed to executing. The blocked state will have a reason associated with it, e.g., blocked-for-I/O-completion, blocked-for-release-event, or blocked-by-cost-overrun.
- *Eligible-for-execution* refers to the state where the schedulable could be selected to have its state changed to executing.

Each type of schedulable defines its own *release events*, for example, the release events for a periodic schedulable are caused by the passage of time and occur at programmatically specified intervals.

Release is the changing of the state of a schedulable from blocked-for-release-event to eligible-for-execution. If the state of a schedulable is blocked-for-release-event when a release event occurs then the state of the schedulable is changed to eligible-for-execution. Otherwise, a state transition from blocked-for-release-event to eligible-for-execution is queued; this is known as a *pending release*. When the next transition of the schedulable into state blocked-for-release-event occurs, and there is a pending release, the state of the schedulable is immediately changed to eligible-for-execution. (Some actions implicitly clear any pending releases.)

Completion is the changing of the state of a schedulable from executing to blocked-for-release-event. Each completion corresponds to a release. A realtime thread is deemed to complete its most recent release when it terminates.

Deadline refers to a time before which a schedulable expects to complete. The i^{th} deadline is associated with the i^{th} release event and a *deadline miss* occurs if the i^{th} completion would occur after the i^{th} deadline.

Deadline monitoring is the process by which the implementation responds to deadline misses. If a deadline miss occurs for a schedulable object, the deadline miss handler, if any, for that schedulable is released. This behaves as if there were an asynchronous event associated with the schedulable, to which the miss handler was bound, and which was fired when the deadline miss occurred.

Periodic, *sporadic*, and *aperiodic* are adjectives applied to schedulables which describe the temporal relationship between consecutive release events. Let R_i denote the time at which a schedulable has had the i^{th} release event occur. Ignoring the effect of release jitter:

- a schedulable is periodic when there exists a value $T > 0$ such that for all i, $R_{i+1} - R_i = T$, where T is called the period;
- a schedulable that is not periodic is said to be aperiodic; and
- an aperiodic schedulable is said to be sporadic when there is a known value $T > 0$ such that for all i, $R_{i+1} - R_i >= T$. T is then called the minimum interarrival time (MIT).

The *cost* of a schedulable is an estimate of the maximum amount of CPU time that the schedulable requires between a release and its associated completion.

The *current CPU consumption* of a schedulable is the amount of CPU time that the schedulable has consumed since its last release.

A *cost overrun* occurs when the schedulable's current CPU consumption becomes greater than, or equal to, its cost.

Cost monitoring is the process by which the implementation tracks CPU consumption and responds to cost overruns. If a cost overrun occurs for a schedulable, the cost overrun handler, if any, for that schedulable is released. This behaves as if there were an asynchronous event associated with the schedulable, to which the overrun handler was bound, and which was fired when the cost overrun occurred.

Cost enforcement is the process by which the implementation ensures that the CPU consumption of a schedulable is no more than the value of the cost parameter in its associated `ReleaseParameters`. (Cost enforcement is an optional facility in an implementation of the `RTSJ`.)

The *base priority* of a schedulable is the priority given in its associated `PriorityParameters` object; the base priority of a Java thread is the priority returned by its `getPriority` method.

When it is not in the enforced state, the *active priority* of a schedulable or a Java thread is the maximum of its base priority and any priority it has acquired due to the action of priority inversion avoidance algorithms (see the *Synchronization Chapter).*

A *processing group* is a collection of schedulables whose combined execution has further execution time constraints which the scheduler uses to govern the group's execution eligibility.

A *scheduler* manages the execution of schedulables: it detects deadline misses and monitors costs. It also manages the execution of Java threads.

The *base scheduler* is an instance of the `PriorityScheduler` class as defined in this specification. This is the initial default scheduler.

A *processor* is 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.

An *affinity* is a set of processors on which the global scheduling of a schedulable can be supported.

6.3 Semantics

This section establishes the semantics that are applicable across the classes of this chapter, and also defines the required scheduling algorithm. Semantics that apply to particular classes, constructors, methods, and fields will be found in the class description and the constructor, method, and field detail sections.

6.3.1 Schedulers

1. Schedulers other than the base scheduler may change the execution eligibility of the schedulables which they manage according to their scheduling algorithm.
2. If an implementation provides any public schedulers other than the base scheduler it shall provide documentation describing each scheduler's semantics in

language and constructs appropriate to the provided scheduling algorithms. This documentation must include the list of classes that constitute schedulables for the scheduler unless that list is the same as the list of schedulables for the base scheduler.

6.3.2 The Base Scheduler

The semantics for the base scheduler assume a uniprocessor or shared memory multiprocessor execution environment. Scheduling is priority preemptive with run to completion, also known as first-in-first-out (FIFO) semantics. The base scheduler supports the execution of all schedulables.

6.3.2.1 Priorities

1. The base scheduler must support at least 28 distinct values (realtime priorities) that can be stored in an instance of `PriorityParameters` in addition to the values 1 through 10 required to support the priorities defined by `java.lang.Thread`. The base priority of each schedulable object under the control of the base scheduler must be from the range of realtime priorities. The realtime priority values must be greater than 10, and they must include all integers from the base scheduler's `getMinPriority()` value to its `getMaxPriority()` value inclusive. The 10 priorities defined for `java.lang.Thread` must effectively have lower execution eligibility than the realtime priorities, but beyond this, their behavior is as defined by the specification of `java.lang.Thread`.
2. Higher priority values in an instance of `PriorityParameters` have a higher execution eligibility.
3. Assignment of any of the realtime priority values to any schedulable controlled by the base priority scheduler is legal. It is the responsibility of application logic to make rational priority assignments.
4. The base scheduler does not use the `importance` value in the `ImportanceParameters` subclass of `PriorityParameters`.
5. For schedulables managed by the base scheduler, the implementation must not change the execution eligibility for any reason other than
   - implementation of a priority inversion avoidance algorithm or
   - as a result of a program's request to change the priority parameters associated with one or more schedulables; e.g., by changing a value in a scheduling parameter object that is used by one or more schedulables, or by using `setSchedulingParameters()` to give a schedulable a different `SchedulingParameters` value.
6. Use of `Thread.setPriority()`, any of the methods defined for schedulables, or any of the methods defined for parameter objects must not affect the cor-

rectness of the priority inversion avoidance algorithms controlled by `PriorityCeilingEmulation` and `PriorityInheritance` — see the *Synchronization* chapter.

7. If schedulable A managed by the base scheduler creates a Java thread, B, then the initial base priority of B is the priority value returned by the `getMaxPriority` method of B's `java.lang.ThreadGroup` object.
8. `PriorityScheduler.getNormPriority()` shall be set to:

```
((PriorityScheduler.getMaxPriority() −
  PriorityScheduler.getMinPriority()) / 3) +
  PriorityScheduler.getMinPriority()
```

9. Hardware priorities, where supported, have values above the base scheduler (see Section 12.3.4).

6.3.2.2 Dispatching

The execution scheduling semantics described in this section are defined in terms of a conceptual model that contains a set of queues of schedulables that are eligible for execution. There is, conceptually, one queue for each priority on each processor. No implementation structures are necessarily implied by the use of this conceptual model. It is assumed that no time elapses during operations described using this model, and therefore no simultaneous operations are possible.

The RTSJ dispatching model specifies its dispatching rules for the default priority scheduler.

1. A schedulable can become a running schedulable only if it is ready and one of the processors in its requested affinity is available.
2. If two schedulables have different active priorities and request the same processor, the schedulable with the higher active priority will always execute in preference to the schedulable with the lower value when both are eligible for execution.
3. Processors are allocated to schedulables based on each schedulable's active priority and their associated affinity.
4. Schedulable dispatching is the process by which one ready schedulable is selected for execution on a processor. This selection is done at certain points during the execution of a schedulable called *schedulable dispatching points*.
5. A schedulable reaches *a schedulable dispatching point* whenever it becomes blocked, when it terminates, or when a higher priority schedulable becomes ready for execution on its processor. That is, a schedulable that is executing will continue to execute until it either blocks, terminates or is preempted by a higher-priority schedulable.

6. The dispatching policy is specified in terms of ready queues and schedulable states. The ready queues are purely conceptual; there is no requirement that such lists physically exist in an implementation. A ready queue is an ordered list of ready schedulable objects. The first position in a queue is called the head of the queue, and the last position is called the tail of the queue.
7. A schedulable is ready if it is in a ready queue, or if it is running. Each processor has one ready queue for each priority value. At any instant, each ready queue of a processor contains exactly the set of schedulables of that priority that are ready for execution on that processor, but are not running on any processor; that is, those schedulables that are ready, are not running on any processor, and can be executed using that processor.
8. A schedulable can be on the ready queues of more than one processor.
9. Each processor has one running schedulable, which is the schedulable currently being executed by that processor. Whenever a schedulable running on a processor reaches a schedulable dispatching point, a new schedulable object is selected to run on that processor. The schedulable selected is the one at the head of the highest priority nonempty ready queue for that processor; this schedulable is then removed from all ready queues to which it belongs.
10. In a multiprocessor system, a schedulable can be on the ready queues of more than one processor. At the extreme, if several processors share the same set of ready schedulables, the contents of their ready queues are identical, and so they can be viewed as sharing one ready queue, and can be implemented that way. Thus, the dispatching model covers multiprocessors where dispatching is implemented using a single ready queue, as well as those with separate dispatching domains.
11. The dispatching mechanism must allow the preemption of the execution of schedulables and Java threads with a bounded delay at a point not governed by the preempted object. The bound on this delay may be implementation-defined, and could be the time to the next point in execution that the heap is in a consistent state or some similar restriction. The implementation should document this bound.
12. A schedulable that is preempted by a higher priority schedulable is placed in the queue for its active priority, at a position determined by the implementation. The implementation must document the algorithm used for such placement. It is recommended that a preempted schedulable be placed at the front of the appropriate queue.
13. A realtime thread that performs a `yield()` is placed at the tail of the queues (dictated by its affinity) for its active priority level.
14. A blocked schedulable that becomes eligible for execution is added to the tail of the queues (dictated by its affinity) for that priority. This behavior also

applies to the initial release of a schedulable.

15. For a schedulable whose active priority is changed as a result of explicitly setting its base priority (through the `PriorityParameters setPriority()` method, the `RealtimeThread setSchedulingParameters()` method, or `Thread`'s `setPriority()` method), this schedulable is added to the tail of the queues (dictated by its affinity) for its new priority level. Queuing when priorities are adjusted by priority inversion avoidance algorithms is governed by semantics specified in the *Synchronization* chapter.

6.3.2.3 Parameter Values

The scheduler uses the values contained in the different parameter objects associated with a schedulable to control the behavior of the schedulable. The scheduler determines what values are valid for the schedulables it manages, which defaults apply and how changes to parameter values are acted upon by the scheduler. Invalid parameter values result in exceptions, as documented in the relevant classes and methods.

1. The default values for the base scheduler are:
   (a) Scheduling parameters are copied from the creating schedulable if possible; if the creating schedulable does not have scheduling parameters, the default is an instance of the default priority parameters value.
   (b) Release parameters default to an instance of the default aperiodic parameters (see `AperiodicParameters`).
   (c) Memory parameters default to null which signifies that memory allocation by the schedulable is not constrained by the scheduler.
   (d) Processing group parameters default to null which signifies that the schedulable is not a member of any processing group and is not subject to processing group based limits on processor utilization.
   (e) The default scheduling parameter values for parameter objects created by a schedulable controlled by the base scheduler are: (see `PriorityScheduler`)

Attribute	Default Value
Priority parameters priority	 norm priority
Importance parameters importance	 No default. A value must be supplied.

2. All numeric or `RelativeTime` attributes in parameter values must be greater than or equal to zero.
3. Values of period must be greater than zero.
4. Deadline values in `ReleaseParameters` objects must be less than or equal to

their period values (where applicable), but the deadline may be greater than the minimum interarrival time in a `SporadicParameters` object.

5. Changes to scheduling, release, memory, and processing group parameters (by methods on the schedulables bound to the parameters or by altering the parameter objects themselves) have the following effect effects:
   (a) They potentially modify the behavior of the scheduler with regard to those schedulables. When such changes in behavior take effect depends on the parameter in question, and the type of schedulable, as described below.
6. Changes to scheduling, release, memory, and processing group parameters are acted upon by the base scheduler as follows:
   (a) Changes to scheduling parameters take effect immediately except when constrained by priority inversion avoidance algorithms.
   (b) Changes to release parameters depend on the parameter being changed, the type of release parameter object and the type of schedulable:
      i. Changes to the deadline and the deadline miss handler take effect at each release event as follows: if the i_{th} release event occurred at a time t_i, then the i^{th} deadline is the time $t_i + D_i$, where D_i is the value of the deadline stored in the schedulable's release parameters object at the time t_i. If a deadline miss occurs then it is the deadline miss handler that was installed in the schedulable's release parameters at time t_i that is released.
      ii. Changes to cost and the cost overrun handler take effect immediately.
      iii. Changes to the period and start time values in `PeriodicParameters` objects are described in "Release of realtime Threads" below.
      iv. Changes to the additional values in `AperiodicParameters` objects and `SporadicParameters` are described, respectively, in "Aperiodic Release Control" and "Sporadic Release Control", below.
      v. Changes to the type of release parameters object generally take effect after completion, except as documented in the following sections.
   (c) Changes to memory parameters take effect immediately.
   (d) Changes to processing group parameters take effect as described in "Processing Groups" below.
   (e) Changes to the scheduler responsible for a schedulable object take effect at completion.

6.3.2.4 Cost Monitoring and Cost Enforcement

The cost of a schedulable is defined by the value returned by invoking the `getCost` method of the schedulable's release parameters object. When a schedulable is initially released it's current CPU consumption is zero and as the schedulable executes, the current CPU consumption increases. For cost monitoring, an implementation must conform to the following requirements.

1. If at any time, due to either execution of the schedulable or a change in the schedulable's cost, the current CPU consumption becomes greater than, or equal to, the current cost of the schedulable, then a cost overrun is triggered. The implementation is required to document the granularity at which the current CPU consumption is updated.
2. When a cost overrun is triggered, the cost overrun handler associated with the schedulable, if any, is released. No further action is taken.
3. The current CPU consumption is reset to zero when the schedulable is next released (i.e. it moves from the blocked-for-release-event state to the eligible-for-execution state).

If cost enforcement is supported, an implementation must conform to the following requirements.

1. When a cost overrun is triggered, in addition to releasing any cost overrun handler, the following actions must be performed.
2. If the most recent release of the schedulable is the i^{th} release, and the $i+1$ release event has not yet occurred, then:
   (a) If the state of the schedulable is either executing or eligible-for-execution, then the schedulable is placed into the state blocked-by-cost-overrun. There may be a bounded delay between the time at which a cost overrun occurs and the time at which the schedulable becomes blocked-by-cost-overrun.
   (b) Otherwise, the schedulable must have been blocked for a reason other than blocked-by-cost-overrun. In this case, the state change to blocked-by-cost-overrun is left pending: if the blocking condition for the schedulable is removed, then its state changes to blocked-by-cost-overrun. There may be a bounded delay between the time at which the blocking condition is removed and the time at which the schedulable becomes blocked-by-cost-overrun.

   Otherwise, if the $i+1$ release event has occurred, the current CPU consumption is set to zero, the schedulable remains in its current state and the cost monitoring system considers the most recent release to now be the $i+1$ release.
3. When the i^{th} release event occurs for a schedulable, the action taken depends on the state of the schedulable:
   (a) If the schedulable is blocked-by-cost-overrun then the cost monitoring system considers the most recent release to be the i^{th} release, the current CPU consumption is set to zero and the schedulable is made eligible for execution;
   (b) Otherwise, if the schedulable is blocked for a reason other than blocked-by-cost-overrun then:

i. If there is a pending state change to blocked-by-cost-overrun then: the pending state change is removed, the cost monitoring system considers the most recent release to be the i^{th} release, the current CPU consumption is set to zero and the schedulable remains in its current blocked state;

ii. Otherwise, no cost monitoring action occurs.

(c) Otherwise no cost monitoring action occurs.

4. When the i^{th} release of a schedulable completes, and the cost monitoring system considers the most recent release to be the i^{th} release, then the current CPU consumption is set to zero and the cost monitoring system considers the most recent release to be the $i + 1$ release. Otherwise, no cost monitoring action occurs.
5. Changes to the cost parameter take effect immediately:
   (a) If the new cost is less than or equal to the current CPU consumption, and the old cost was greater than the current CPU consumption, then a cost overrun is triggered.
   (b) If the new cost is greater than the current CPU consumption:
      i. If the schedulable is blocked-by-cost-overrun, then the schedulable is made eligible for execution;
      ii. Otherwise, if the schedulable is blocked for a reason other than blocked-by-cost-overrun, and there is a pending state change to blocked-by-cost-overrun, then the pending state change is removed;
      iii. Otherwise, no cost monitoring action occurs.
6. The state of the cost monitoring system for a schedulable can be *reset* by the scheduler (see 6e in the Release of realtime Threads section, below). If the most recent release of the schedulable is considered to be the m^{th} release, and the most recent release event for the schedulable was the n^{th} release event (where $n > m$), then a reset causes the cost monitoring system to consider the most recent release to be the n^{th} release, and to zero the current CPU consumption.

6.3.2.5 Release of Realtime Threads

The repeated release of realtime threads is achieved by executing in a loop and invoking the `RealtimeThread.waitForNextPeriod` or `RealtimeThread.waitForNextRelease` methods, or their interruptible equivalents (`RealtimeThread.waitForNextPeriodInterruptible` `RealtimeThread.waitForNextReleaseInterruptible`) within that loop. For simplicity, unless otherwise stated, the semantics in this section apply to both forms of those methods.

1. A realtime thread's release characteristics are determined by the following:
   (a) The invocation of the realtime thread's `start` method and the value of

its phasing policy parameter (if applicable).

(b) The action of the RealtimeThread methods: waitForNextPeriod, waitForNextPeriodInterruptible, schedulePeriodic, deschedulePeriodic, waitForNextRelease, waitForNextReleaseInterruptible, schedule, and deschedule;

(c) The occurrence of deadline misses and whether or not a miss handler is installed; and

(d) The passing of time that generates periodic release events and a call of the release method that generates aperiodic release events.

2. The *initial release event* of a periodic realtime thread occurs in response to the invocation of its start method in accordance with the start time specified in its release parameters and its assigned phasing policy — see PeriodicParameters and PhasingPolicy. The *initial release event* of an aperiodic realtime thread occurs immediately in response to the invocation of the its start method.
3. Changes to the start time in a realtime thread's PeriodicParameters object only have an effect on its initial release time. Consequently, if a PeriodicParameters object is bound to multiple realtime threads, a change in the start time may affect all, some or none, of those threads, depending on whether or not start has been invoked on them.
4. Subsequent release events occur
   (a) for periodic realtime threads: when each period falls due, except as described below (in 6e), at times determined as follows: if the i^{th} release event occurred at a time t_i, then the $i+1$ release event occurs at the time $t_i + T_i$, where T_i is the value of the period stored in the realtime thread's PeriodicParameters object at the time t_i.
   (b) for aperiodic realtime thread: with each call of the release method, except as described below (in 6e)
   (c) for sporadic realtime threads: with each call of the release method, except as described below (in 6e) with additional regulation to enforce MIT are required as defined in *Sporadic Release Control* below.
5. Each release of an aperiodic realtime thread is an arrival. If the thread has release parameters of type AperiodicParameters, then the arrival may become a release event for the thread according to the semantics given in "Aperiodic Release Control" below. If the thread has release parameters of type SporadicParameters, then the arrival may become a release event for the thread according to the semantics given in "Sporadic Release Control" below. If the thread has release parameters of a type other than SporadicParameters then the arrival is a release event, and the arrival-time is the release event time.
6. The implementation should behave effectively as if the following state variables were added to a realtime thread's state,

boolean `descheduled`,
integer `pendingReleases`,
integer `missCount`, and
boolean `lastReturn`.

and manipulated by the actions as described below:

(a) Initially:
    `descheduled` = false,
    `pendingReleases` = 0,
    `missCount` = 0, and
    `lastReturn` = true.
(b) When the realtime thread's `deschedulePeriodic` or `deschedule` method is invoked: set the value of `descheduled` to true.
(c) When the realtime thread's `schedulePeriodic` or `schedule` method is invoked: set the value of `descheduled` to false; then if the thread is blocked-for-release-event, set the value of `pendingReleases` to zero, and tell the cost monitoring and enforcement system to reset for this thread.
(d) When `descheduled` is true, the realtime thread is said to be *descheduled*.
(e) A realtime thread that has been descheduled and is blocked-for-release-event will not receive any further release events until after it has been rescheduled by a call to `schedulePeriodic` or `schedule`; this means that no deadline misses can occur until the thread has been rescheduled. The descheduling of a realtime thread has no effect on its initial release.
(f) When each release event occurs:
    i. If the state of the realtime thread is blocked-for-release-event (that is, it is waiting in `waitForNextPeriod` or `waitForNextRelease`), then if the thread is descheduled then do nothing, else increment the value of `pendingReleases`, inform cost monitoring and enforcement that the next release event has occurred, and notify the thread to make it eligible for execution;
    ii. Otherwise, increment the value of `pendingReleases`, and inform cost monitoring and enforcement that the next release event has occurred.
(g) On each deadline miss:
    i. If the realtime thread has a deadline miss handler: set the value of `descheduled` to true, atomically release the handler with its `fireCount` increased by the value of `missCount + 1` and zero `missCount`;
    ii. Otherwise add one to the `missCount` value.
(h) When the `waitForNextPeriod` or `waitForNextRelease` method is invoked by the current realtime thread there are two possible behaviors depending on the value of `missCount`:
    i. If `missCount` is greater than zero: decrement the `missCount` value; then if the `lastReturn` value is false, completion occurs: apply any

pending parameter changes, decrement `pendingReleases`, inform cost monitoring and enforcement the realtime thread has completed and return false; otherwise set the `lastReturn` value to false and return false.

ii. Otherwise, apply any pending parameter changes, inform cost monitoring and enforcement of completion, and then wait while `descheduled` is true, or `pendingReleases` is zero. Then set the `lastReturn` value to true, decrement `pendingReleases`, and return true.

7. An invocation of the `waitForNextPeriodInterruptible` or `waitForNextReleaseInterruptible`method behaves as described above with the following additions:
   (a) If the invocation commences when an instance of `AsynchronouslyInterruptedException` (AIE) is pending on the realtime thread, then the invocation immediately completes abruptly by throwing that pending instance as an `InterruptedException`. If this occurs, the most recent release has not completed. If the pending instance is the generic AIE instance then the interrupt state of the realtime thread is cleared.
   (b) If an instance of AIE becomes pending on the realtime thread while it is blocked-for-release-event, and the realtime thread is descheduled, then the AIE remains pending until the realtime thread is no longer descheduled. The associated reschedule acts as a release event. Execution then continues as in (d) where the time value used as t_{int} is the time at which the schedulable was rescheduled.
   (c) If an instance of AIE becomes pending on the realtime thread while it is blocked-for-release-event, and it is not descheduled, then this acts as a release event. Execution the continues as in (d) where the time value used as t_{int} is the time at which the AIE becomes pending.
   (d) i. The realtime thread is made eligible for execution.
      ii. Upon execution, the invocation completes abruptly by throwing the pending AIE instance as an `InterruptedException`. If the pending instance is the generic AIE instance then the interrupt state of the realtime thread is cleared.
      iii. The deadline associated with this release is the time $t_{int}+D_{int}$, where D_{int} is the value of the deadline stored in the realtime thread's release parameters object at the time t_{int}.
      iv. The next release time for the realtime thread will be $t_{int}+T_{int}$, where T_{int} is the value of the period stored in the realtime thread's release parameters object at the time t_{int}.
      v. Cost monitoring and enforcement is informed of the release event.

When the thrown AIE instance is caught, the AIE becomes pending again (as per the usual semantics for AIE) until it is explicitly cleared.

8. Changes to release parameter types are treated as a pseudo RE-START of the realtime thread and
   (a) any old pending releases are cleared
   (b) any old arrival queue is flushed
   (c) any outstanding call to deschedule is cleared
   (d) any outstanding deadline misses are cleared

   The semantics are described below:
   (a) Effect on the realtime thread if it is not waiting for next release event (and is not descheduled)
      i. no effect until the end of current release
      ii. when the change occurs it is a pseudo re-start of the thread. i.e. if new parameters are aperiodic — the release is immediate; if new parameters are periodic — the periodic start time algorithm is used.
   (b) Effect on the realtime thread if it is not waiting for next release event (but there is an outstanding descheduled).
      i. there is an immediate "schedule" of the thread
      ii. there is no further effect until end of current release
      iii. when change occurs it is a pseudo re-start of the thread, i.e. if new parameters are aperiodic — the release is immediate; if new parameters are periodic — the periodic start time algorithm is used.
   (c) Effect on the realtime thread if it is waiting for next release event (and not descheduled)
      i. From Periodic to Aperiodic — when the next periodic release event occurs, the thread becomes aperiodic with an immediate release
      ii. From Aperiodic to Periodic — there is an immediate pseudo re-start of the thread using the periodic start time algorithm
   (d) Effect on realtime thread if waiting for next release event (but there is an outstanding descheduled)
      i. there is an immediate "schedule" of the thread
      ii. From Periodic to Aperiodic — when the next periodic release event occurs, the thread becomes aperiodic with an immediate release
      iii. From Aperiodic to Periodic — there is an immediate pseudo re-start of the thread using the periodic start time algorithm

6.3.2.5.1 Pseudo-Code for Periodic and Aperiodic Thread Actions The semantics of the previous section can be more clearly understood by viewing them in pseudo-code form for each of the methods and actions involved. In the following no mechanism for blocking and unblocking a thread is prescribed. The use of the wait and notify terminology in places is purely an aid to expressing the desired semantics in familiar terms.

```
// These values are part of thread state.
boolean descheduled = false;
int pendingReleases = 0;
boolean lastReturn = true;
int missCount = 0;
int currentRP;
int newRP;
int periodic = 1;
int aperiodic = 2;
int sporadic = 3;
boolean RPchange = false;
boolean started = false; // set to true on first release;

changeReleaseParameters(int newP)
{
  newRP = newP;

  descheduled = false; // automatic re−schedule
  if (blocked−for−release−event)
  {
    if (currentRP == periodic)
    {
      // defer until next release
      RPChange = true;
    }
    else
    {
      // immediate change; current is aperiodic or sporadic
      performParameterChanges();
      assert pendingReleases = 0
      assert missCount = 0;
      started = false; // flush arrival queue
      costMonitoringReset();
      currentRP = newRP;
      if (newRP == periodic)
      {
        // consider this as the equivalent of call the
        // start method of the RT thread.
        // If start time has passed, generate a
        // an "onNextPeriodDue" event.
        // Otherwise, arrange for the event to be
        // generate at the appropriate time
      }
      else
      {
        // aperiodic or sporadic
        // generate a releaseArrivalEvent
```

```
      }
    }
  }
  else
  {
    // not at end of release, defer change
    RPChange = true;
  }
}

schedulePeriodic()
{
  descheduled = false;
  if (blocked−for−release−event)
  {
    pendingReleases = 0; // flush arrival time queue
    costMonitoringReset();
  }
}

deschedulePeriodic()
{
  if (!RPChange started)
  {
    // no deschedule if outstanding RPchange
    // or not started
    descheduled = true;
  }
}

schedule()
{
  descheduled = false;
  if (blocked−for−release−event)
  {
    pendingReleases = 0; // flush arrival time queue
    costMonitoringReset();
  }
}

deschedule()
{
  if (!RPChange started)
  {
    // no deschedule if outstanding RPchange
    // or not started
    descheduled = true;
  }
}
```

```

onAperiodicReleaseArrival()
{
  if (!started) started = true;
  if (currentRP == periodic) throw IllegalThreadStateException;
  if (descheduled)
  {
    ; // do nothing
  }
  else
  {
    perform\_any\_execution\_regulation
    // For a sporadic thread, the onReleaseDue event
    // will be generated when MIT concerns have been satisfied
    // For an aperiodic thread, this will
    // immediately generate an onReleaseDue event.
  }
}

onAperiodicReleaseDue()
{
  if (currentRP == periodic) throw panic;
  if (blocked−for−release−event)
  {
    if (descheduled)
    {
      ; // do nothing
    }
    else
    {
      pendingReleases++;
      notifyCostMonitoringOfReleaseEvent()
      notify it; // make eligible for execution
    }
  }
  else
  {
    pendingReleases++;
    notifyCostMonitoringOfReleaseEvent();
  }
}

onNextPeriodeDue()
{
  // also called on first release
  if (!started) started = true;
  if (currentRP != periodic) panic;
  if (blocked−for−release−event)
  {
```

```
    if (descheduled)
    {
      ; // do nothing
    }
    else
    {
      pendingReleases++;
      notifyCostMonitoringOfReleaseEvent();
      notify it; // make eligible for execution
    }
  }
  else
  {
    pendingReleases++;
    notifyCostMonitoringOfReleaseEvent();
  }
}

onDeadlineMiss()
{
  if (there is a miss handler)
  {
    descheduled = true;
    release miss handler with fireCount increased by missCount+1
    missCount = 0;
  }
  else
  {
    missCount++;
  }
}

waitForNextRelease()
{
  assert(pendingReleases \& = 0);
  if (missCount > 0 )
  {
    // Missed a deadline without a miss handler
    missCount−−;
    if (lastReturn == false)
    {
      // Changes on completion take place here
      performParameterChanges()
      notifyCostMonitoringOfCompletion();
      if (RPchange)
      {
        RPChangeNow();
        return true;
      }
```

```
      else
      {
        pendingReleases--;
      }
    }
    lastReturn = false;
    return false;
  }
  else
  {
    // Changes on completion take place here
    performParameterChanges();
    notifyCostMonitoringOfCompletion();
    if (RPchange)
    {
      RPChangeNow();
      return True;
    }
    wait while (descheduled || pendingReleases == 0);
    // blocked-for-release-event
    // check again for RP change
    if (RPchange)
    {
      RPChangeNow();
    }
    pendingReleases--;
    lastReturn = true;
    return true;
  }
}

waitForNextPeriod
{
  assert(pendingReleases >= 0);
  if (missCount > 0 )
  {
    // Missed a deadline without a miss handler
    missCount--;
    if (lastReturn == false)
    {
      // Changes "on completion" take place here
      performParameterChanges();
      pendingReleases--;
      notifyCostMonitoringOfCompletion();
    }
    lastReturn = false;
    return false;
  }
  else
```

```
  {
    // Changes "on completion" take place here
    performParameterChanges();
    notifyCostMonitoringOfCompletion();
    if (RPchange)
    {
      RPChangeNow();
      return True;
    }
    wait while (descheduled || pendingReleases == 0);
    // blocked-for-release-event
    pendingReleases--;
    lastReturn = true;
    return true;
  }
}


changeRPNow()
{
  // Changing over RP
  // Assuming clean slate!
  RPchange = false;
  pendingReleases = 0;
  flushArrivalQueue();
  // this removes all outstanding releases
  missCount = 0;
  // restart here
  if (newRP == periodic)
  {
    // consider this as the equivalent of call the
    // start method of the RT thread
    if !start time has passed
    {
      // arrange for timing event to be generated
      started = false;
      currentRP = newRP;
      wait while (pendingReleases == 0);
      // blocked-for-release-event
    }
  }
  else
  {
    // aperiodic or sporadic
    // record a releaseArrivalEvent
  }
  started = true;
  currentRP = newRP;
  lastReturn = true;
```

```
}
```

6.3.2.6 Aperiodic Release Control

Aperiodic schedulables are released in response to events occurring, such as the starting of a realtime thread, the calling of the `release` method of a realtime thread, or the firing of an associated asynchronous event for an asynchronous event handler. The occurrence of these events, each of which is a potential release event, is termed an *arrival*, and the time that they occur is termed the *arrival time*.

The base scheduler behaves effectively as if it maintained a queue, called the arrival time queue, for each aperiodic schedulable object. This queue maintains information related to each release event (including any parameters passed with the release mechanism) from its "arrival" time until the associated release completes, or another release event occurs — whichever is later. If an arrival is accepted into the arrival time queue, then it is a release event and the time of the release event is the arrival time. The initial size of this queue is an attribute of the schedulable's aperiodic parameters, and is set when an aperiodic parameter object is first associated with the schedulable. Over time the queue may become full and its behavior in this situation is determined by the queue overflow policy specified in the schedulable's aperiodic parameters. There are four overflow policies defined:

Policy	Action on Overflow
`IGNORE`	Silently ignore the arrival. The arrival is not accepted, no release event occurs, and, if the arrival was caused programmatically (such as by invoking `fire` on an asynchronous event), the caller is not informed that the arrival has been ignored.
`EXCEPT`	Throw an `ArrivalTimeQueueOverflowException`. The arrival is not accepted, and no release event occurs, but if the arrival was caused programmatically, the caller will have `ArrivalTimeQueueOverflowException` thrown.
`REPLACE`	The arrival is not accepted and no release event occurs. If the completion associated with the last release event in the queue has not yet occurred, and the deadline has not been missed, then the release event time for that release event is replaced with the arrival time of the new arrival and any associated parameters overwritten. This will alter the deadline for that release event. If the completion associated with the last release event has occurred, or the deadline has already been missed, then the behavior of the `REPLACE` policy is equivalent to the `IGNORE` policy.
`SAVE`	Behave effectively as if the queue were expanded as necessary to accommodate the new arrival. The arrival is accepted and a release event occurs.

Under the SAVE policy the queue can grow and shrink over time.

Changes to the queue overflow policy take effect immediately. When an arrival occurs and the queue is full, the policy applied is the policy as defined at that time.

6.3.2.7 Sporadic Release Control

Sporadic parameters include a minimum interarrival time, MIT, that characterizes the expected frequency of releases. When an arrival is accepted implementation behaves as if it calculates the earliest time at which the next arrival could be accepted, by adding the current MIT to the arrival time of this accepted arrival. The scheduler guarantees that each sporadic schedulable it manages, is released at most once in any MIT. It implements two mechanisms for enforcing this rule:

1. *Arrival-time regulation* controls the work-load by considering the time between arrivals. If a new arrival occurs earlier than the expected next arrival time then a MIT violation has occurred, and the scheduler acts to prevent a release from occurring that would break the “one release per MIT” guarantee. Three arrival-time MIT-violation policies are supported:

Policy	Action on Violation
IGNORE	Silently ignore the violating arrival. The arrival is not accepted, no release event occurs, and, if the arrival was caused programmatically (such as by invoking `fire` on an asynchronous event), the caller is not informed that the arrival has been ignored.
EXCEPT	Throw a `MITViolationException`. The arrival is not accepted, and no release event occurs, but if the arrival was caused programmatically, the caller will have `MITViolationException` thrown.
REPLACE	The arrival is not accepted and no release event occurs. If the completion associated with the last release event in the queue has not yet occurred, and the deadline has not been missed, then the release event time for that release event is replaced with the arrival time of the new arrival and any associated parameters overwritten. This will alter the deadline for that release event. If the completion associated with the last release event has occurred, or the deadline has already been missed, then the behavior of the REPLACE policy is equivalent to the IGNORE policy.

2. *Execution-time regulation* occurs if the MIT violation policy SAVE is in effect. Under this policy all arrivals are accepted, but the scheduler behaves effectively as if released schedulable objects were further constrained by a scheduling policy that restricts execution to at most one release per MIT. This policy is only able to delay the effective release of a schedulable. The deadline of each release event is always set relative to its arrival time. This policy may not schedule the effective release of an async event handler until after its deadline has passed. In this case the deadline miss handler is released at the deadline time even though the related async event has not yet reached its effective release.
   The SAVE policy makes no direct use of the next expected arrival time, but it maintains the value in case the MIT violation policy is changed from SAVE to one of the arrival-time regulation policies.

The *effective release time* of a release event *i* is the earliest time that the handler can be released in response to that release event. It is determined for each release event based on the MIT policy in force at the release event time:

1. For IGNORE, EXCEPT and REPLACE the effective release time is the release event time.
2. For SAVE the effective release time of release event *i* is the effective release time of release event *i-1* plus the current value of the MIT.

The scheduler will delay the release associated with the release event at the head of the arrival time queue until the current time is greater than or equal to the effective release time of that release event.

Changes to minimum interarrival time and the MIT violation policy take effect immediately, but only affect the next expected arrival time, and effective release time, for release events that occur after the change.

6.3.2.8 Release Control for Asynchronous Event Handlers

Asynchronous event handlers can be associated with one or more asynchronous events. When an asynchronous event is fired, all handlers associated with it are released, according to the semantics below:

1. Each firing of an associated asynchronous event is an arrival. If the handler has release parameters of type `AperiodicParameters`, then the arrival may become a release event for the handler, according to the semantics given in "Aperiodic Release Control" above. If the handler has release parameters of type `SporadicParameters`, then the arrival may become a release event for the handler, according to the semantics given in "Sporadic Release Control" above. If the handler has release parameters of a type other than `SporadicParameters` then the arrival is a release event, and the arrival-time is the release event time.
2. For each release event that occurs for a handler, an entry is made in the arrival-time queue and the handler's `fireCount` is incremented by one.
3. Initially a handler is considered to be blocked-for-release-event and its `fireCount` is zero.
4. Releases of a handler are serialized by having its `handleAsyncEvent` method invoked repeatedly while its `fireCount` is greater than zero:
   (a) Before invoking handleAsyncEvent, the `fireCount` is decremented and the front entry (if still present) removed from the arrival-time queue.
   (b) Each invocation of `handleAsyncEvent`, in this way, is a release.
   (c) The return from `handleAsyncEvent` is the completion of a release.
   (d) Processing of any exceptions thrown by `handleAsyncEvent` occurs prior to completion.
5. The deadline for a release is relative to the release event time and determined at the release event time according to the value of the deadline contained in the handler's release parameters. This value does not change, except as described previously for handlers using a REPLACE policy for MIT violation or arrival-time queue overflow.
6. The application code can directly modify the `fireCount` as follows:
   (a) The `getAndDecrementPendingFireCount` method decreases the `fireCount` by one (if it was greater than zero), and returns the old value.

This removes the front entry from the arrival-time queue but otherwise has no effect on the scheduling of the current schedulable, nor the handler itself. Any data parameter passed with the associated fire request is lost.

(b) The `getAndClearPendingFireCount` method is functionally equivalent to invoking `getAndDecrementPendingFireCount` until it returns zero, and returning the original `fireCount` value. Any data parameters passed with the associated fire requests are lost.

7. The scheduler may delay the invocation of `handleAsyncEvent` to ensure the effective release time honors any restrictions imposed by the MIT violation policy, if applicable, of that release event.
8. Cost monitoring and enforcement for an asynchronous event handler interacts with release events and completions as previously defined with the added requirement that at the completion of `handleAsyncEvent`, if the `fireCount` is now zero, then the cost monitoring and enforcement system is told to reset for this handler.

6.3.2.9 Processing Groups

A processing group is defined by a processing group parameters object, and each schedulable that is bound to that parameter object is called a *member* of that processing group. A processing group has an associated affinity that contains only one processor.

Processing groups are only functional in a system that implements processing group enforcement. Although the processing group itself does not consume CPU time, it acts as a proxy for its members.

6.3.2.9.1 Definitions for Processing Groups The *enforced priority* of a schedulable is a priority with no execution eligibility.

6.3.2.9.2 Semantics for Processing Groups

1. The deadline of a processing group is defined by the value returned by invoking the `getDeadline` method of the processing group parameters object.
2. A deadline miss for the processing group is triggered if any member of the processing group consumes CPU time at a time greater than the deadline for the most recent release of the processing group.
3. When a processing group misses a deadline:
   (a) If the processing group has a miss handler, it is released for execution
   (b) If the processing group has no miss handler, no action is taken.
4. The cost of a processing group is defined by the value returned by invoking the `getCost` method of the processing group parameters object.

5. When a processing group is initially released, its current CPU consumption is zero and as the members of the processing group execute, the current CPU consumption increases. The current CPU consumption is set to zero in response to certain actions as described below.
6. If at any time, due to either execution of the members of the processing group or a change in the parameter group's cost, the current CPU consumption becomes greater than, or equal to, the current cost of the processing group, then a cost overrun is triggered. The implementation is required to document the granularity at which the current CPU consumption is updated.
7. When a cost overrun is triggered, the cost overrun handler associated with the processing group, if any, is released, and the processing group enters the `enforced state`. For each member of the processing group:
   (a) The schedulable is placed into the enforced state.
   (b) When a schedulable is in the enforced state the base scheduler schedules that schedulable effectively as if the enforced priority were used in place of the schedulable's base priority.
8. When the release event occurs for a processing group, the action taken depends on the state of the processing group:
   (a) If the processing group is not in the enforced state then the current CPU consumption for the group is set to zero;
   (b) Otherwise the processing group is in the enforced state. It is removed from the enforced state, the current CPU consumption of the group is set to zero, and each member of the group is removed from the enforced state.
9. Changes to the cost parameter take effect immediately:
   (a) If the new cost is less than or equal to the current CPU consumption, and the old cost was greater than the current CPU consumption, then a cost overrun is triggered.
   (b) If the new cost is greater than the current CPU consumption:
      i. If the processing group is enforced, then the processing group behaves as defined in semantic 8.
      ii. Otherwise, no cost monitoring and enforcement action occurs.
10. Changes to other parameters take place as follows:
   (a) Start: can only be changed before the parameters group is started; i.e., before the start time or before the parameter object is associated with any schedulable. Changes take effect immediately.
   (b) Period: at each release the next period is set based on the current value of the processing group's period.
   (c) Deadline: at each release the next deadline is set based on the current value of the processing group's deadline.

   (d) OverrunHandler: at each release the overrunHandler is set based on the current value of the processing group's overrunHandler.
   (e) MissHandler: at each release the missHandler is set based on the current value of the processing group's missHandler.
11. Changes to the membership of the processing group take effect immediately.
12. The start time for the processing group may be relative or absolute.
   (a) If the start time is absolute, the processing group behaves effectively as if the initial release time were the start time.
   (b) If the start time is relative, the initial release time is computed relative to the time `start` or `fire` (as appropriate) is first called for a member of the processing group.

Note: Until a processing group starts, its budget cannot be replenished, but its members will be enforced if they exceed the initial budget. Also, once a processing group is started it behaves effectively as if it continued running continuously until the defining `ProcessingGroupParameters` object is freed.

6.4 Package javax.realtime

6.4.1 Interfaces

6.4.1.1 BoundSchedulable

A marker interface to provide a type safe reference to all schedulables that are bound to a single underlying thread. A `RealtimeThread`[1] is by definition bound.

Interfaces

Schedulable

6.4.1.2 Schedulable

Handlers and other objects can be dispatched by a `Scheduler`[2] when they provide a `run()` method and the methods defined below. The `Scheduler`[3] uses this information to create a suitable context to execute the `run()` method.

Interfaces

`Runnable`
Timable

6.4.1.2.1 Methods

---

getMemoryParameters

Gets a reference to the `MemoryParameters`[4] object for this schedulable.

Signature

```
public
javax.realtime.MemoryParameters getMemoryParameters()
```

Returns

[1] Section 5.3.2.2
[2] Section 6.4.2.9
[3] Section 6.4.2.9
[4] Section 11.4.3.7

A reference to the current MemoryParameters[5] object.

getProcessingGroupParameters

Gets a reference to the ProcessingGroupParameters[6] object for this schedulable.

Signature

```
public
javax.realtime.ProcessingGroupParameters
getProcessingGroupParameters()
```

Returns

A reference to the current ProcessingGroupParameters[7] object.

getConfigurationParameters

Gets a reference to the ConfigurationParameters[8] object for this schedulable.
Available since RTSJ 2.0

Signature

```
public
javax.realtime.ConfigurationParameters
getConfigurationParameters()
```

Returns

A reference to the associated ConfigurationParameters[9] object.

getReleaseParameters

Gets a reference to the ReleaseParameters[10] object for this schedulable.

[5] Section 11.4.3.7
[6] Section 6.4.2.7
[7] Section 6.4.2.7
[8] Section 5.3.2.1
[9] Section 5.3.2.1
[10] Section 6.4.2.8

Signature

```
public
javax.realtime.ReleaseParameters getReleaseParameters()
```

Returns

A reference to the current `ReleaseParameters`[11] object.

getScheduler

Gets a reference to the `Scheduler`[12] object for this schedulable.

Signature

```
public
javax.realtime.Scheduler getScheduler()
```

Returns

A reference to the associated `Scheduler`[13] object.

getSchedulingParameters

Gets a reference to the `SchedulingParameters`[14] object for this schedulable.

Signature

```
public
javax.realtime.SchedulingParameters getSchedulingParameters()
```

Returns

A reference to the current `SchedulingParameters`[15] object.

setMemoryParameters(MemoryParameters)

Sets the memory parameters associated with this instance of `Schedulable`.

This change becomes effective under conditions determined by the scheduler controlling the schedulable. For instance, the change may be immediate or it may be delayed until the next release of the schedulable object. See the documentation for the scheduler for details.

[11] Section 6.4.2.8
[12] Section 6.4.2.9
[13] Section 6.4.2.9
[14] Section 6.4.2.10
[15] Section 6.4.2.10

Signature

```
public
void setMemoryParameters(MemoryParameters memory)
```

Parameters

memory A `MemoryParameters`[16] object which will become the memory parameters associated with `this` after the method call. When `null`, the default value is governed by the associated scheduler (a new object is created when the default value is not `null`). (See `PriorityScheduler`[17].)

Throws

IllegalArgumentException when `memory` is not compatible with the schedulable's scheduler. Also when this schedulable is no-heap and `memory` is located in heap memory.

IllegalAssignmentError when the schedulable cannot hold a reference to `memory`, or when `memory` cannot hold a reference to this schedulable instance.

IllegalThreadStateException when the schedulable's scheduler prohibits this parameter change at this time due to the state of the schedulable.

setProcessingGroupParameters(ProcessingGroupParameters)

Sets the `ProcessingGroupParameters`[18] of `this`.

This change becomes effective under conditions determined by the scheduler controlling the schedulable. For instance, the change may be immediate or it may be delayed until the next release of the schedulable. See the documentation for the scheduler for details.

Signature

```
public
void setProcessingGroupParameters(ProcessingGroupParameters
group)
```

Parameters

group A `ProcessingGroupParameters`[19] object which will take effect as determined by the associated scheduler. When `null`, the default value is governed

[16] Section 11.4.3.7
[17] Section 6.4.2.6
[18] Section 6.4.2.7
[19] Section 6.4.2.7

by the associated scheduler (a new object is created when the default value is not `null`). (See `PriorityScheduler`[20].)

Throws

IllegalArgumentException Thrown when `group` is not compatible with the scheduler for this schedulable object. Also when this schedulable is no-heap and `group` is located in heap memory.

IllegalAssignmentError when `this` object cannot hold a reference to `group` or `group` cannot hold a reference to `this`.

IllegalThreadStateException when the schedulable's scheduler prohibits the changing of the processing group parameter at this time due to the state of the schedulable object.

setReleaseParameters(ReleaseParameters)

Sets the release parameters associated with this instance of `Schedulable`.

This change becomes effective under conditions determined by the scheduler controlling the schedulable. For instance, the change may be immediate or it may be delayed until the next release of the schedulable. The different properties of the release parameters may take effect at different times. See the documentation for the scheduler for details.

Signature

```
public
void setReleaseParameters(ReleaseParameters release)
```

Parameters

release A `ReleaseParameters`[21] object which will become the release parameters associated with this after the method call, and take effect as determined by the associated scheduler. When `null`, the default value is governed by the associated scheduler (a new object is created when the default value is not `null`). (See `PriorityScheduler`[22].)

Throws

IllegalArgumentException Thrown when `release` is not compatible with the associated scheduler. Also when this schedulable is no-heap and `release` is located in heap memory.

IllegalAssignmentError when `this` object cannot hold a reference to `release` or `release` cannot hold a reference to `this`.

[20] Section 6.4.2.6
[21] Section 6.4.2.8
[22] Section 6.4.2.6

IllegalThreadStateException when the schedulable's scheduler prohibits the changing of the release parameter at this time due to the state of the schedulable.

setScheduler(Scheduler)

Sets the reference to the Scheduler object. The timing of the change must be agreed between the scheduler currently associated with this schedulable, and `scheduler`.

Signature

```
public
void setScheduler(Scheduler scheduler)
```

Parameters

scheduler A reference to the scheduler that will manage execution of this schedulable. `Null` is not a permissible value.

Throws

IllegalArgumentException Thrown when `scheduler` is `null`, or the schedulable's existing parameter values are not compatible with `scheduler`. Also when this schedulable is no-heap and `scheduler` is located in heap memory.

IllegalAssignmentError when the schedulable cannot hold a reference to `scheduler`.

SecurityException when the caller is not permitted to set the scheduler for this schedulable.

IllegalThreadStateException when `scheduler` refuses to accept this schedulable at this time due to the state of the schedulable.

setScheduler(Scheduler, SchedulingParameters, ReleaseParameters, MemoryParameters, ProcessingGroupParameters)

Sets the scheduler and associated parameter objects. The timing of the change must be agreed between the scheduler currently associated with this schedulable, and `scheduler`.

Signature

```
public
void setScheduler(Scheduler scheduler, SchedulingParameters
scheduling, ReleaseParameters release, MemoryParameters
```

`memoryParameters, ProcessingGroupParameters group)`

Parameters

scheduler A reference to the scheduler that will manage the execution of this schedulable. `Null` is not a permissible value.

scheduling A reference to the `SchedulingParameters`[23] which will be associated with `this`. When `null`, the default value is governed by `scheduler` (a new object is created when the default value is not `null`). (See `PriorityScheduler`[24].)

release A reference to the `ReleaseParameters`[25] which will be associated with `this`. When `null`, the default value is governed by `scheduler` (a new object is created when the default value is not `null`). (See `PriorityScheduler`[26].)

memoryParameters A reference to the `MemoryParameters`[27] which will be associated with `this`. When `null`, the default value is governed by `scheduler` (a new object is created when the default value is not `null`). (See `PriorityScheduler`[28].)

group A reference to the `ProcessingGroupParameters`[29] which will be associated with `this`. When `null`, the default value is governed by `scheduler` (a new object is created). (See `PriorityScheduler`[30].)

Throws

IllegalArgumentException Thrown when `scheduler` is `null` or the parameter values are not compatible with `scheduler`. Also thrown when this schedulable is no-heap and `scheduler`, `scheduling` `release`, `memoryParameters`, or `group` is located in heap memory.

IllegalAssignmentError when `this` object cannot hold references to all the parameter objects or the parameters cannot hold references to `this`.

IllegalThreadStateException when `scheduler` prohibits the changing of the scheduler or a parameter at this time due to the state of the schedulable.

SecurityException when the caller is not permitted to set the scheduler for this schedulable.

setSchedulingParameters(SchedulingParameters)

Sets the scheduling parameters associated with this instance of `Schedulable`.

[23]Section 6.4.2.10
[24]Section 6.4.2.6
[25]Section 6.4.2.8
[26]Section 6.4.2.6
[27]Section 11.4.3.7
[28]Section 6.4.2.6
[29]Section 6.4.2.7
[30]Section 6.4.2.6

This change becomes effective under conditions determined by the scheduler controlling the schedulable. For instance, the change may be immediate or it may be delayed until the next release of the schedulable. See the documentation for the scheduler for details.

Signature

```
public
void setSchedulingParameters(SchedulingParameters scheduling)
```

Parameters

scheduling A reference to the `SchedulingParameters`[31] object. When `null`, the default value is governed by the associated scheduler (a new object is created when the default value is not `null`). (See `PriorityScheduler`[32].)

Throws

IllegalArgumentException Thrown when `scheduling` is not compatible with the associated scheduler. Also when this schedulable is no-heap and `scheduling` is located in heap memory.

IllegalAssignmentError when `this` object cannot hold a reference to `scheduling` or `scheduling` cannot hold a reference to `this`.

IllegalThreadStateException when the schedulable's scheduler prohibits the changing of the scheduling parameter at this time due to the state of the schedulable object.

getMinConsumption(RelativeTime)

Available since RTSJ 2.0

Signature

```
public
javax.realtime.RelativeTime getMinConsumption(RelativeTime dest)
```

Returns

The minimum CPU consumption for this schedulable in any single release. When this method is called on the current schedulable, the CPU consumption of the current release is not considered. When `dest` is `null`, return the minimum consumption in a `RelativeTime`[33] instance from the current allocation context. When `dest` is not `null`, return the minimum consumption in `dest`

[31] Section 6.4.2.10
[32] Section 6.4.2.6
[33] Section 9.4.1.3

getMinConsumption

Equivalent to `getMinConsumption(null)`.
Available since RTSJ 2.0

Signature

```
public
javax.realtime.RelativeTime getMinConsumption()
```

getMaxConsumption(RelativeTime)

Available since RTSJ 2.0

Signature

```
public
javax.realtime.RelativeTime getMaxConsumption(RelativeTime dest)
```

Returns

The maximum CPU consumption for this schedulable in any single release. When this method is called on the current schedulable, the CPU consumption of the current release is not considered. When `dest` is `null`, return the maximum consumption in a `RelativeTime`[34] instance from the current allocation context. When `dest` is not `null`, return the maximum consumption in `dest`

getMaxConsumption

Equivalent to `getMaxConsumption(null)`.
Available since RTSJ 2.0

Signature

```
public
javax.realtime.RelativeTime getMaxConsumption()
```

[34]Section 9.4.1.3

wakeUp

Provides a means for a `Clock`[35] to end a sleep.
Available since RTSJ 2.0

Signature

```
public
void wakeUp()
```

Throws

IllegalStateException when called from user code.

6.4.2 Classes

6.4.2.1 Affinity

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

An affinity set is a set of processors that can be associated with a `Thread` (including realtime threads), async event handler, active event dispatchers and processing group parameters instance. For instances of `Thread` and async event handlers, the associated affinity set binds the `Thread` or async event handler to the set of processors.

Each implementation supports an array of predefined affinity sets. They can be used either to reflect the scheduling arrangement of the underlying OS or they can be used by the system designer to impose defaults for, say, Java threads, non-heap realtime schedulables etc. A program is only allowed to dynamically create new affinity sets with cardinality of one. This restriction reflects the concern that not all operating systems will support multiprocessor affinity sets.

The processor membership of an affinity set is immutable. The Java thread, schedulable, and `ProcessingGroupParameters`[36] associations of an affinity set are mutable. The processor affinity of instances of `Thread` and bound async event handlers can be changed by static methods in this class. The processor affinity

[35]Section 10.4.2.1
[36]Section 6.4.2.7

of un-bound asnc event handlers is fixed to a default value, as returned by the `getHeapDefault()`[37] and `getNoHeapDefault()`[38] methods.

The internal representation of a set of processors in an `Affinity` instance is not specified, but the representation that is used to communicate with this class is a `BitSet` where each bit corresponds to a logical processor ID. The relationship between logical and physical processors is beyond the scope of this specification, and may change.

The affinity set factory only generates usable `Affinity` instances; i.e., affinity sets that (at least when they are created) can be used with `set(Affinity, BoundAsyncEventHandler)`[39], `set(Affinity, Thread)`[40], and `set(Affinity, ProcessingGroupParameters)`[41]. The factory cannot create an affinity set with more than one processor member, but such affinity sets are supported. They may be internally created by the RTSJ runtime at startup time.

The set of affinity sets created at startup (the predefined set) is visible through the `getPredefinedAffinities(Affinity[])`[42] method.

The affinity set factory may be used to create affinity sets with a single processor member at any time, though this operation only supports processor members that are valid as the processor affinity for a thread (at the time of the affinity set's creation.)

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

Real-time threads and bound async event handlers that have processing group parameters are members of that processing group, and their processor affinity is governed by the intersection of the processing group's affinity and the schedulable's affinity. The affinity set of a processing group must have exactly one processor. The intersection of a non-default PG affinity set with the schedulable's affinity set must contain at most one entry. When the intersection is empty, the affinity defaults.

Ordinarily, an execution context inherits its creator's affinity set, but

- Java threads do not inherit affinity from `Schedulable`[43].
- Instance of `AsyncEventHandler`[44] that are not bound cannot be assigned a non-default affinity.

[37]Section 6.4.2.1.2
[38]Section 6.4.2.1.2
[39]Section 6.4.2.1.2
[40]Section 6.4.2.1.2
[41]Section 6.4.2.1.2
[42]Section 6.4.2.1.2
[43]Section 6.4.1.2
[44]Section 8.4.3.5

- A `Schedulable` does not inherit affinity from Java threads.

When an execution context does not inherit its creator's affinity set, its initial affinity set defaults as specified in this class:

- The default used when a heap-using schedulable does not inherit its creator's affinity set, and for all unbound heap-using async event handlers.
- The default used when a no-heap schedulable does not inherit its creator's affinity set, and for all unbound no-heap async event handlers.
- The default used for Java threads created by schedulables.

This class also controls the default affinity used when a processing group is created. That value is the set of all available processors. (Which permits each member of the processing group to use the affinity set it would use if it were in no processing group.) The processor affinity of the processing group can subsequently be altered with the {`set(Affinity, ProcessingGroupParameters)`[45] method.

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

Available since RTSJ 2.0

Inheritance

java.lang.Object
  javax.realtime.Affinity

6.4.2.1.1 Constructors

6.4.2.1.2 Methods

---

generate(BitSet)

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

Platforms that support specific affinity sets will register those `Affinity` instances with `Affinity`[46]. They appear in the arrays returned by `getPredefinedAffinities()`[47] and `getPredefinedAffinities(Affinity[])`[48].

[45]Section 6.4.2.1.2
[46]Section 6.4.2.1
[47]Section 6.4.2.1.2
[48]Section 6.4.2.1.2

Signature

```
public static final
javax.realtime.Affinity generate(BitSet bitSet)
```

Parameters

bitSet The `BitSet` associated with the generated `Affinity`.

Throws

NullPointerException when `bitSet` is `null`.

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

Returns

The resulting `Affinity`.

get(BoundSchedulable)

Return the affinity set instance associated with `handler`.

Signature

```
public static final
javax.realtime.Affinity get(BoundSchedulable schedulable)
```

Parameters

handler a bound async event handler.

Returns

The associated affinity set.

get(Thread)

Return the affinity set instance associated with `thread`.

Signature

```
public static final
javax.realtime.Affinity get(Thread thread)
```

Parameters

thread a Java thread, or one of its subclasses (including `RealtimeThread`[49]).

Returns

The associated affinity set.

get(ProcessingGroupParameters)

Return the affinity set instance associated with `pgp`.

Signature

```
public static final
javax.realtime.Affinity get(ProcessingGroupParameters pgp)
```

Parameters

pgp An instance of `ProcessingGroupParameters`[50]

Returns

The associated affinity set.

get(javax.realtime.ActiveEventDispatcher)

Return the affinity set instance associated with `dispatcher`.

Signature

```
public static final
javax.realtime.Affinity get(javax.realtime.ActiveEventDispatcher
dispatcher)
```

Parameters

dispatcher An instance of `ActiveEventDispatcher`[51]

Returns

The associated affinity set.

getAvailableProcessors

This method is equivalent to `getAvailableProcessors(BitSet)`[52] with a `null` argument.

[49]Section 5.3.2.2
[50]Section 6.4.2.7
[51]Section 8.4.3.3
[52]Section 6.4.2.1.2

Signature

```
public static final
java.util.BitSet getAvailableProcessors()
```

Returns

the set of processors available to the program.

getAvailableProcessors(BitSet)

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

Signature

```
public static final
java.util.BitSet getAvailableProcessors(BitSet dest)
```

Parameters

dest When `dest` is non-null, use `dest` as the returned value. When it is `null`, create a new `BitSet`.

Returns

A `BitSet` representing the set of processors currently valid for use in the `bitset` argument to `generate(BitSet)`[53].

getHeapDefault

Return the default processor affinity set for heap-using schedulables.

Signature

```
public static final
javax.realtime.Affinity getHeapDefault()
```

Returns

The current default processor affinity set for heap-using schedulables.

[53]Section 6.4.2.1.2

getDefault

Return the default processor affinity for Java threads.

Signature

```
public static final
javax.realtime.Affinity getDefault()
```

Returns

The current default processor affinity set for Java threads.

getNoHeapDefault

Return the default processor affinity for non-heap mode schedulable objects.

Signature

```
public static final
javax.realtime.Affinity getNoHeapDefault()
```

Returns

The current default processor affinity set for non-heap mode schedulables.

getProcessingGroupDefault

Return the processor affinity set used for SOs where the intersection of their affinity set and their processing group parameters' affinity set yields the empty set.

Signature

```
public static final
javax.realtime.Affinity getProcessingGroupDefault()
```

Returns

The affinity set associated with schedulables for which the intersection of their affinity and their `ProcessingGroupParameters`[54] affinity would be the empty set.

getPredefinedAffinitiesCount

[54] Section 6.4.2.7

Return the minimum array size required to store references to all the predefined processor affinity sets.

Signature

```
public static final
int getPredefinedAffinitiesCount()
```

Returns

The minimum array size required to store references to all the predefined affinity sets.

getPredefinedAffinities

Equivalent to invoking `getPredefinedAffinitySets(null)`.

Signature

```
public static final
javax.realtime.Affinity[] getPredefinedAffinities()
```

Returns

an array of the pre-defined affinity sets.

getPredefinedAffinities(javax.realtime.Affinity[])

Return an array containing all affinity sets that were predefined by the Java runtime.

Signature

```
public static final
javax.realtime.Affinity[]
getPredefinedAffinities(javax.realtime.Affinity[] dest)
```

Parameters

dest The destination array, or `null`.

Throws

IllegalArgumentException when `dest` is not large enough.

Returns

`dest` or a newly created array when `dest` is `null`, populated with references to the pre-defined affinity sets.

When `dest` has excess entries, they are filled with `null`.

isSetAffinitySupported

Determine whether or not affinity control is supported.

Signature

```
public static final
boolean isSetAffinitySupported()
```

Returns

`true` when the `set(Affinity, Thread)`[55] family of methods is supported.

set(Affinity, BoundAsyncEventHandler)

Set the processor affinity of a bound AEH to `set`.

Signature

```
public static final
void set(Affinity set, BoundAsyncEventHandler aeh)
throws ProcessorAffinityException
```

Parameters

set The processor affinity set

aeh The bound async event handler

Throws

ProcessorAffinityException Thrown when the runtime fails to set the affinity for platform-specific reasons.

NullPointerException when `set` or `aeh` is `null`.

set(Affinity, Thread)

Set the processor affinity of a Java thread or `RealtimeThread`[56] to `set`.

Signature

```
public static final
void set(Affinity set, Thread thread)
throws ProcessorAffinityException
```

Parameters

[55] Section 6.4.2.1.2

[56] Section 5.3.2.2

set The processor affinity set
thread The thread or real-time thread.

Throws

ProcessorAffinityException when the runtime fails to set the affinity for platform-specific reasons.
NullPointerException when `set` or `thread` is `null`.

set(Affinity, ProcessingGroupParameters)

Set the processor affinity of `pgp` to `set`.

Signature

```
public static final
void set(Affinity set, ProcessingGroupParameters pgp)
throws ProcessorAffinityException
```

Parameters

set The processor affinity set
pgp The processing group parameters instance.

Throws

ProcessorAffinityException when the runtime fails to set the affinity for platform-specific reasons or `pgp` contains more than one processor.
NullPointerException when `set` or `pgp` is `null`.

set(Affinity, javax.realtime.ActiveEventDispatcher)

Set the processor affinity of `dispatcher` to `set`.

Signature

```
public static final
void set(Affinity set, javax.realtime.ActiveEventDispatcher
dispatcher)
throws ProcessorAffinityException
```

Parameters

set The processor affinity set

Throws

ProcessorAffinityException when the runtime fails to set the affinity for platform-specific reasons or `dispatcher` contains more than one processor.
NullPointerException when `set` or `dispatcher` is `null`.

getProcessors

Return a `BitSet` representing the processor affinity set for this `Affinity`.

Signature

```
public final
java.util.BitSet getProcessors()
```

Returns

A newly created `BitSet` representing this `Affinity`.

getProcessors(BitSet)

Return a `BitSet` representing the processor affinity set of this `Affinity`.
Available since RTSJ 2.0

Signature

```
public final
java.util.BitSet getProcessors(BitSet dest)
```

Parameters

dest Set `dest` to the `BitSet` value. When `dest` is `null`, create a new `BitSet` in the current allocation context.

Returns

A `BitSet` representing the processor affinity set of this `Affinity`.

isProcessorInSet(int)

Ask whether a processor is included in this affinity set.
Available since RTSJ 2.0

Signature

```
public final
boolean isProcessorInSet(int processorNumber)
```

Parameters

processorNumber

Returns

True when and only when `processorNumber` is represented in this affinity set.

applyTo(BoundAsyncEventHandler)

Set the processor affinity of a bound AEH to `this`.

Signature

```
public final
void applyTo(BoundAsyncEventHandler aeh)
throws ProcessorAffinityException
```

Parameters

aeh The bound async event handler

Throws

ProcessorAffinityException Thrown when the runtime fails to set the affinity for platform-specific reasons.
NullPointerException when `aeh` is `null`.

applyTo(Thread)

Set the processor affinity of a Java thread or `RealtimeThread`[57] to `this`.

Signature

```
public final
void applyTo(Thread thread)
throws ProcessorAffinityException
```

Parameters

thread The thread or realtime thread.

Throws

ProcessorAffinityException when the runtime fails to set the affinity for platform-specific reasons.
NullPointerException when `thread` is `null`.

applyTo(ProcessingGroupParameters)

Set the processor affinity of `pgp` to `this`.

[57] Section 5.3.2.2

Signature

```
public final
void applyTo(ProcessingGroupParameters pgp)
throws ProcessorAffinityException
```

Parameters

pgp The processing group parameters instance.

Throws

ProcessorAffinityException when the runtime fails to set the affinity for platform-specific reasons or `pgp` contains more than one processor.

NullPointerException when `pgp` is `null`.

applyTo(javax.realtime.ActiveEventDispatcher)

Set the processor affinity of `dispatcher` to `this`.

Signature

```
public final
void applyTo(javax.realtime.ActiveEventDispatcher dispatcher)
throws ProcessorAffinityException
```

Parameters

dispatcher is the dispatcher instance.

Throws

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

NullPointerException when `dispatcher` is `null`.

6.4.2.2 AperiodicParameters

When a reference to an `AperiodicParameters` object is given as a parameter to a schedulable's constructor or passed as an argument to one of the schedulable's setter methods, the `AperiodicParameters` object becomes the release parameters object bound to that schedulable. Changes to the values in the `AperiodicParameters` object affect that schedulable. When bound to more than one schedulable, changes to the values in the `AperiodicParameters` object affect *all* of the associated objects. Note that this is a one-to-many relationship and *not* a many-to-many.

Only changes to an `AperiodicParameters` object caused by methods on that object cause the change to propagate to all schedulables using the object. For instance, calling `setCost` on an `AperiodicParameters` object will make the change, then notify that the scheduler that the parameter object has changed. At that point the object is reconsidered for every schedulable that uses it. Invoking a method on the `RelativeTime` object that is the cost for this object may change the cost but it does not pass the change to the scheduler at that time. That change must not change the behavior of the schedulable's that use the parameter object until a setter method on the `AperiodicParameters` object is invoked, or the parameter object is used in `setReleaseParameters()` or a constructor for a schedulable.

The implementation must use modified copy semantics for each `HighResolutionTime`[58] parameter value. The value of each time object should be treated as if it were copied at the time it is passed to the parameter object, but the object reference must also be retained. For instance, the value returned by `getCost()` must be the same object passed in by setCost(), but any changes made to the time value of the cost must not take effect in the associated `AperiodicParameters` instance unless they are passed to the parameter object again, e.g. with a new invocation of `setCost`.

Correct initiation of the deadline miss and cost overrun handlers requires that the underlying system know the arrival time of each sporadic task. For an instance of `RealtimeThread`[59] the arrival time is the time at which the `start()` is invoked. For other instances of `Schedulable`[60], the required behaviors may require the implementation to behave effectively as if it maintained a queue of arrival times.

The following tables gives the default values for the constructors parameters.

Attribute	Value
cost	`new RelativeTime(0,0)`
deadline	`new RelativeTime(Long.MAX_VALUE, 999999)`
overrunHandler	None
missHandler	None
Arrival time queue size	0
Queue overflow policy	SAVE

Caution: This class is explicitly unsafe for multithreading when being changed. Code that mutates instances of this class should synchronize at a higher level.

Inheritance

[58] Section 9.4.1.2
[59] Section 5.3.2.2
[60] Section 6.4.1.2

java.lang.Object
javax.realtime.ReleaseParameters
javax.realtime.AperiodicParameters

6.4.2.2.1 Fields

---

arrivalTimeQueueOverflowExcept

`public static final arrivalTimeQueueOverflowExcept`

Represents the "EXCEPT" policy for dealing with arrival time queue overflow. Under this policy, when an arrival occurs and its time should be queued but the queue already holds a number of times equal to the initial queue length defined by `this` then the `fire()` method shall throw a `ArrivalTimeQueueOverflowException`[61]. Any other associated semantics are governed by the schedulers for the schedulables using these aperiodic parameters. When the arrival is a result of a happening to which the instance of `AsyncEventHandler`[62] is bound then the arrival time is ignored.
Available since RTSJ 1.0.1 Moved here from `SporadicParameters`.

arrivalTimeQueueOverflowIgnore

`public static final arrivalTimeQueueOverflowIgnore`

Represents the "IGNORE" policy for dealing with arrival time queue overflow. Under this policy, when an arrival occurs and its time should be queued, but the queue already holds a number of times equal to the initial queue length defined by `this` then the arrival is ignored. Any other associated semantics are governed by the schedulers for the schedulables using these aperiodic parameters.
Available since RTSJ 1.0.1 Moved here from `SporadicParameters`.

arrivalTimeQueueOverflowReplace

`public static final arrivalTimeQueueOverflowReplace`

[61] Section 14.2.2.1
[62] Section 8.4.3.5

Represents the "REPLACE" policy for dealing with arrival time queue overflow. Under this policy, when an arrival occurs and should be queued but the queue already holds a number of times equal to the initial queue length defined by `this` then the information for this arrival replaces a previous arrival. Any other associated semantics are governed by the schedulers for the schedulables using these aperiodic parameters.
Available since RTSJ 1.0.1 Moved here from `SporadicParameters`.

arrivalTimeQueueOverflowSave

```
public static final arrivalTimeQueueOverflowSave
```

Represents the "SAVE" policy for dealing with arrival time queue overflow. Under this policy, when an arrival occurs and should be queued but the queue is full, then the queue is lengthened and the arrival time is saved. Any other associated semantics are governed by the schedulers for the schedulables using these aperiodic parameters. This policy does not update the "initial queue length" as it alters the actual queue length. Since the `SAVE` policy grows the arrival time queue as necessary, for the `SAVE` policy the initial queue length is only an optimization.
Available since RTSJ 1.0.1 Moved here from `SporadicParameters`.

6.4.2.2.2 Constructors

AperiodicParameters

Create an `AperiodicParameters` object. This constructor is equivalent to

```
AperiodicParameters(null, null, null, null)
```

Available since RTSJ 1.0.1

Signature

```
public
  AperiodicParameters()
```

AperiodicParameters(RelativeTime)

Create an `AperiodicParameters` object. This constructor is equivalent to

```
AperiodicParameters(null, deadline, null, null)
```

Available since RTSJ 2.0

Signature

```
public
  AperiodicParameters(RelativeTime deadline)
```

Parameters

deadline

AperiodicParameters(RelativeTime, AsyncEventHandler)

Create an `AperiodicParameters` object. This constructor is equivalent to:

```
AperiodicParameters(null, deadline, null, missHandler)
```

Available since RTSJ 2.0

Signature

```
public
  AperiodicParameters(RelativeTime deadline, AsyncEventHandler missHandler)
```

AperiodicParameters(RelativeTime, RelativeTime, AsyncEventHandler, AsyncEventHandler)

Create an `AperiodicParameters` object.

Available since RTSJ 2.0

Signature

```
public
  AperiodicParameters(RelativeTime cost, RelativeTime deadline, AsyncEventHandl
```

Parameters

cost Processing time per invocation. On implementations which can measure the amount of time a schedulable object is executed, this value is the maximum amount of time a schedulable receives. On implementations which cannot measure execution time, it is not possible to determine when any particular object exceeds cost. When `null`, the default value is a new instance of `RelativeTime(0,0)`.

deadline The latest permissible completion time measured from the release time of the associated invocation of the schedulable. When `null`, the default value is a new instance of `RelativeTime(Long.MAX_VALUE, 999999)`.

overrunHandler This handler is invoked when an invocation of the schedulable exceeds cost. Not required for minimum implementation. When `null`, the default value is no overrun handler.

missHandler This handler is invoked when the `run()` method of the schedulable object is still executing after the deadline has passed. When `null`, the default value is no miss handler.

Throws

IllegalArgumentException when the time value of `cost` is less than zero, or the time value of `deadline` is less than or equal to zero.

IllegalAssignmentError when `cost`, `deadline`, `overrunHandler` or `missHandler` cannot be stored in `this`.

6.4.2.2.3 Methods

---

getArrivalTimeQueueOverflowBehavior

Gets the behavior of the arrival time queue in the event of an overflow.
Available since RTSJ 1.0.1 Moved from `SporadicParameters`

Signature

```
public
java.lang.String getArrivalTimeQueueOverflowBehavior()
```

Returns

The behavior of the arrival time queue as a string.

getInitialArrivalTimeQueueLength

Gets the initial number of elements the arrival time queue can hold. This returns the initial queue length currently associated with this parameter object. When the overflow policy is `SAVE` the initial queue length may not be related to the current queue lengths of schedulables associated with this parameter object.
Available since RTSJ 1.0.1 Moved here from `SporadicParameters`.

Signature

```
public
int getInitialArrivalTimeQueueLength()
```

Returns

The initial length of the queue.

setDeadline(RelativeTime)

Sets the deadline value.

When this parameter object is associated with any schedulable object (by being passed through the schedulable's constructor or set with a method such as `RealtimeThread.setReleaseParameters(ReleaseParameters)`[63]) the deadline of those schedulables is altered as specified by each schedulable's respective scheduler.

Signature

```
public
void setDeadline(RelativeTime deadline)
```

Parameters

deadline The latest permissible completion time measured from the release time of the associated invocation of the schedulable. When `deadline` is `null`, the deadline is set to a new instance of `RelativeTime(Long.MAX_VALUE, 999999)`.

Throws

IllegalArgumentException when the time value of `deadline` is less than or equal to zero, or when the new value of this deadline is incompatible with the scheduler for any associated schedulable.

IllegalAssignmentError @inheritDoc

[63]Section 5.3.2.2.2

setArrivalTimeQueueOverflowBehavior(String)

Sets the behavior of the arrival time queue in the case where the insertion of a new element would make the queue size greater than the initial size given in this.

Values of `behavior` are compared using reference equality (==) not value equality (`equals()`).

Available since RTSJ 1.0.1 Moved here from `SporadicParameters`.

Signature

```
public
void setArrivalTimeQueueOverflowBehavior(String behavior)
```

Parameters

behavior A string representing the behavior.

Throws

IllegalArgumentException when `behavior` is not one of the `final` queue overflow behavior values defined in this class.

setInitialArrivalTimeQueueLength(int)

Sets the initial number of elements the arrival time queue can hold without lengthening the queue. The initial length of an arrival queue is set when the schedulable using the queue is constructed, after that time changes in the initial queue length are ignored.

Available since RTSJ 1.0.1 Moved here from `SporadicParameters`.

Signature

```
public
void setInitialArrivalTimeQueueLength(int initial)
```

Parameters

initial The initial length of the queue.

Throws

IllegalArgumentException when `initial` is less than zero.

6.4.2.3 ImportanceParameters

Importance is an additional scheduling metric that may be used by some priority-based scheduling algorithms during overload conditions to differentiate execution order among threads of the same priority.

In some realtime systems an external physical process determines the period of many threads. When rate-monotonic priority assignment is used to assign priorities, many of the threads in the system may have the same priority because their periods are the same. However, it is conceivable that some threads may be more important than others and in an overload situation importance can help the scheduler decide which threads to execute first. The base scheduling algorithm represented by `PriorityScheduler`[64] must not consider importance.

Inheritance

java.lang.Object
  javax.realtime.SchedulingParameters
    javax.realtime.PriorityParameters
      javax.realtime.ImportanceParameters

6.4.2.3.1 Constructors

ImportanceParameters(int, int)

Create an instance of `ImportanceParameters`.

Signature

```
public
  ImportanceParameters(int priority, int importance)
```

Parameters

priority The priority value assigned to schedulables that use this parameter instance. This value is used in place of the value passed to `Thread.setPriority`.
importance The importance value assigned to schedulable objects that use this parameter instance.

6.4.2.3.2 Methods

---

[64]Section 6.4.2.6

getImportance

Gets the importance value.

Signature

```
public
int getImportance()
```

Returns

The value of importance for the associated instances of `Schedulable`[65].

setImportance(int)

Set the importance value. When this parameter object is associated with any schedulable (by being passed through the schedulable's constructor or set with a method such as `RealtimeThread.setSchedulingParameters(SchedulingParameters)`[66]) the importance of those schedulables is altered at a moment controlled by the schedulers for the respective schedulables.

Signature

```
public
void setImportance(int importance)
```

Parameters

importance The value to which importance is set.

Throws

IllegalArgumentException when the given importance value is incompatible with the scheduler for any of the schedulables which are presently using this parameter object.

toString

Print the value of the priority and importance values of the associated instance of `Schedulable`[67]

Signature

[65]Section 6.4.1.2
[66]Section 5.3.2.2.2
[67]Section 6.4.1.2

```
public
java.lang.String toString()
```

6.4.2.4 PeriodicParameters

This release parameter indicates that the schedulable is released on a regular basis. For an `AsyncEventHandler`[68], this means that the handler is either released by a periodic timer, or the associated event occurs periodically. For a `RealtimeThread`[69], this means that the `RealtimeThread.waitForNextPeriod`[70] or `RealtimeThread.waitForNextPeriodInterruptible`[71] method will unblock the associated realtime thread at the start of each period.

When a reference to a `PeriodicParameters` object is given as a parameter to a schedulable's constructor or passed as an argument to one of the schedulable's setter methods, the `PeriodicParameters` object becomes the release parameters object bound to that schedulable. Changes to the values in the `PeriodicParameters` object affect that schedulable object. When bound to more than one schedulable then changes to the values in the `PeriodicParameters` object affect *all* of the associated objects. Note that this is a one-to-many relationship and *not* a many-to-many.

Only changes to a `PeriodicParameters` object caused by methods on that object cause the change to propagate to all schedulable objects using the object. For instance, calling `setCost` on an `PeriodicParameters` object will make the change, then notify that the scheduler that the parameter object has changed. At that point the object is reconsidered for every SO that uses it. Invoking a method on the `RelativeTime` object that is the cost for this object may change the cost but it does not pass the change to the scheduler at that time. That change must not change the behavior of the SOs that use the parameter object until a setter method on the `PeriodicParameters` object is invoked, or the parameter object is used in `setReleaseParameters()` or a constructor for an SO.

Periodic parameters use `HighResolutionTime`[72] values for period and start time. Since these times are expressed as a `HighResolutionTime`[73] values, these values use accurate timers with nanosecond granularity. The actual resolution available and even the quantity the timers measure depend on the clock associated with each time

[68] Section 8.4.3.5
[69] Section 5.3.2.2
[70] Section 5.3.2.2.2
[71] Section 5.3.2.2.2
[72] Section 9.4.1.2
[73] Section 9.4.1.2

value.

The implementation must use modified copy semantics for each `HighResolutionTime`[74] parameter value. The value of each time object should be treated as if it were copied at the time it is passed to the parameter object, but the object reference must also be retained. For instance, the value returned by `getCost()` must be the same object passed in by setCost(), but any changes made to the time value of the cost must not take effect in the associated `PeriodicParameters` instance unless they are passed to the parameter object again, e.g. with a new invocation of `setCost`.

The following table gives the default parameter values for the constructors.

Attribute	Default Value
start	new RelativeTime(0,0)
period	No default. A value must be supplied
cost	new RelativeTime(0,0)
deadline	new RelativeTime(period)
overrunHandler	None
missHandler	None

Periodic release parameters are strictly informational when they are applied to async event handlers. They must be used for any feasibility analysis, but release of the async event handler is not entirely controlled by the scheduler.

Caution: This class is explicitly unsafe for multithreading when being changed. Code that mutates instances of this class should synchronize at a higher level.

Inheritance

java.lang.Object
  javax.realtime.ReleaseParameters
    javax.realtime.PeriodicParameters

6.4.2.4.1 Constructors

PeriodicParameters(RelativeTime)

[74] Section 9.4.1.2

Create a `PeriodicParameters` object with the specified period and all other attributes set to their default values. This constructor has the same effect as invoking `PeriodicParameters(null, period, null, null, null, null)`
Available since RTSJ 1.0.1

Signature

```
public
  PeriodicParameters(RelativeTime period)
```

PeriodicParameters(javax.realtime.HighResolutionTime<?>, RelativeTime)

Create a `PeriodicParameters` object with the specified period and start times, and all other attributes set to their default values. This constructor has the same effect as invoking `PeriodicParameters(start, period, null, null, null, null)`
Available since RTSJ 1.0.1

Signature

```
public
  PeriodicParameters(javax.realtime.HighResolutionTime<?> start, RelativeTime perio
```

PeriodicParameters(javax.realtime.HighResolutionTime<?>, RelativeTime, RelativeTime)

Create a `PeriodicParameters` object with the specified deadline, period and start times, and all other attributes set to their default values. This constructor has the same effect as invoking `PeriodicParameters(start, period, null, deadline, null, null, null)`
Available since RTSJ 2.0

Signature

```
public
  PeriodicParameters(javax.realtime.HighResolutionTime<?> start, RelativeTime p
```

PeriodicParameters(javax.realtime.HighResolutionTime<?>, RelativeTime, RelativeTime, RelativeTime, AsyncEventHandler, AsyncEventHandler)

Create a `PeriodicParameters` object with attributes set to the specified values.

Signature

```
public
  PeriodicParameters(javax.realtime.HighResolutionTime<?> start, RelativeTime p
```

Parameters

start Time at which the first release begins (i.e. the real-time thread becomes eligible for execution.) When a `RelativeTime`, this time is relative to the first time the thread becomes activated (that is, when `start()` is called). When an `AbsoluteTime`, then the first release is the maximum of the start parameter and the time of the call to the associated RealtimeThread.start() method (modified according to any phasing policy). When null, the default value is a new instance of `RelativeTime(0,0)`.

period The period is the interval between successive unblocks of the `RealtimeThread.waitForNextPeriod`[75] and `RealtimeThread.waitForNextPeriodInterruptible`[76] methods. There is no default value. When `period` is null an exception is thrown.

cost Processing time per release. On implementations which can measure the amount of time a schedulable is executed, this value is the maximum amount of time a schedulable receives per release. When null, the default value is a new instance of `RelativeTime(0,0)`.

deadline The latest permissible completion time measured from the release time of the associated invocation of the schedulable. When null, the default value is new instance of `RelativeTime(period)`.

overrunHandler This handler is invoked when an invocation of the schedulable exceeds cost in the given release. Implementations may ignore this parameter. When null, the default value is no overrun handler.

[75]Section 5.3.2.2.2
[76]Section 5.3.2.2.2

missHandler This handler is invoked when the `run()` method of the schedulable is still executing after the deadline has passed. When null, the default value is no deadline miss handler.

Throws

IllegalArgumentException when the `period` is null or its time value is not greater than zero, or when the time value of `cost` is less than zero, or when the time value of `deadline` is not greater than zero, or when the clock associated with the `cost` is not the real-time clock, or when the clock associated with the `start`, `deadline` and `period` parameters are not the same.

IllegalAssignmentError when `start period`, `cost`, `deadline`, `overrunHandler` or `missHandler` cannot be stored in `this`.

6.4.2.4.2 Methods

---

getPeriod

Gets the period.

Signature

```
public
javax.realtime.RelativeTime getPeriod()
```

Returns

The current value in `period`.

getStart

Gets the start time.

Signature

```
public
javax.realtime.HighResolutionTime<?> getStart()
```

Returns

The current value in `start`. This is the value that was specified in the constructor or by `setStart()`, not the actual absolute time corresponding to the start of one of the schedulable objects associated with this `PeriodicParameters` object.

setDeadline(RelativeTime)

Sets the deadline value.

When this parameter object is associated with any schedulable object (by being passed through the schedulable's constructor or set with a method such as RealtimeThread.setReleaseParameters(ReleaseParameters)[77]) the deadline of those schedulables is altered as specified by each schedulable's respective scheduler.

Signature

```
public
void setDeadline(RelativeTime deadline)
```

Parameters

deadline The latest permissible completion time measured from the release time of the associated invocation of the schedulable. When `deadline` is `null`, the deadline is set to a new instance of `RelativeTime` equal to `period`.

Throws

IllegalArgumentException when the time value of `deadline` is less than or equal to zero, or when the new value of this deadline is incompatible with the scheduler for any associated schedulable.

IllegalAssignmentError @inheritDoc

setPeriod(RelativeTime)

Sets the period.

Signature

```
public
void setPeriod(RelativeTime period)
```

Parameters

period The value to which `period` is set.

Throws

IllegalArgumentException when the given period is `null` or its time value is not greater than zero. Also when `period` is incompatible with the scheduler for any associated schedulable or when an associated `AbstractAsyncEventHandler`[78] is associated with a `Timer`[79] whose period does not match `period`.

[77]Section 5.3.2.2.2
[78]Section 8.4.3.2
[79]Section 10.4.2.5

IllegalAssignmentError when `period` cannot be stored in `this`.

setStart(javax.realtime.HighResolutionTime<?>)

Sets the start time.

The effect of changing the start time for any schedulables associated with this parameter object is determined by the scheduler associated with each schedulable.

Note: An instance of `PeriodicParameters` may be shared by several schedulables. A change to the start time may take effect on a subset of these schedulables. That leaves the start time returned by `getStart` unreliable as a way to determine the start time of a schedulable.

Signature

```
public
void setStart(javax.realtime.HighResolutionTime<?> start)
```

Parameters

start The new start time. When `null`, the default value is a new instance of `RelativeTime(0,0)`.

Throws

IllegalArgumentException when the given start time is incompatible with the scheduler for any of the schedulable objects which are presently using this parameter object.

IllegalAssignmentError when `start` cannot be stored in `this`.

6.4.2.5 PriorityParameters

Instances of this class should be assigned to schedulables that are managed by schedulers which use a single integer to determine execution order. The base scheduler required by this specification and represented by the class `PriorityScheduler`[80] is such a scheduler.

Inheritance

java.lang.Object
  javax.realtime.SchedulingParameters
    javax.realtime.PriorityParameters

[80] Section 6.4.2.6

6.4.2.5.1 Constructors

PriorityParameters(int)

Create an instance of PriorityParameters[81] with the given priority.

Signature

```
public
  PriorityParameters(int priority)
```

Parameters

priority The priority assigned to schedulables that use this parameter instance.

6.4.2.5.2 Methods

getPriority

Gets the priority value.

Signature

```
public
int getPriority()
```

Returns

The priority.

setPriority(int)

Set the priority value. When this parameter object is associated with any schedulable (by being passed through the schedulable's constructor or set with a method such as RealtimeThread.setSchedulingParameters(SchedulingParameters)[82])

[81]Section 6.4.2.5
[82]Section 5.3.2.2.2

the base priority of those schedulables is altered as specified by each schedulable's scheduler.

Signature

```
public
void setPriority(int priority)
```

Parameters

priority The value to which priority is set.

Throws

IllegalArgumentException when the given priority value is incompatible with the scheduler for any of the schedulables which are presently using this parameter object.

toString

Converts the priority value to a string.

Signature

```
public
java.lang.String toString()
```

Returns

A string representing the value of priority.

6.4.2.6 PriorityScheduler

Class which represents the required (by the RTSJ) priority-based scheduler. The default instance is the base scheduler which does fixed priority, preemptive scheduling.

This scheduler, like all schedulers, governs the default values for scheduling-related parameters in its client schedulables. The defaults are as follows:

Attribute	Default Value
Priority parameters	
Priority	norm priority

Note that the system contains one instance of the `PriorityScheduler` which is the system's base scheduler and is returned by `PriorityScheduler.instance()`. It may, however, contain instances of subclasses of `PriorityScheduler` and even additional instances of `PriorityScheduler` itself created through this class' protected constructor. The instance returned by the `instance()`[83] method is the *base scheduler* and is returned by `Scheduler.getDefaultScheduler()`[84] unless the default scheduler is reset with `Scheduler.setDefaultScheduler(Scheduler)`[85].

Inheritance

java.lang.Object
  javax.realtime.Scheduler
    javax.realtime.PriorityScheduler

6.4.2.6.1 Fields

MAX_PRIORITY

`public static final MAX_PRIORITY`

The maximum priority value used by the implementation.

Deprecated since RTSJ version as of RTSJ 1.0.1 Use the `getMaxPriority`[86] method instead.

MIN_PRIORITY

`public static final MIN_PRIORITY`

The minimum priority value used by the implementation.

Deprecated since RTSJ version as of RTSJ 1.0.1 Use the `getMinPriority`[87] method instead.

6.4.2.6.2 Constructors

---

[83] Section 6.4.2.6.3
[84] Section 6.4.2.9.2
[85] Section 6.4.2.9.2
[86] Section 6.4.2.6.3
[87] Section 6.4.2.6.3

PriorityScheduler

Construct an instance of `PriorityScheduler`. Applications will likely not need any instance other than the default instance.

Signature

```
protected
  PriorityScheduler()
```

6.4.2.6.3 Methods

instance

Return a reference to the distinguished instance of `PriorityScheduler` which is the system's base scheduler.

Signature

```
public static
javax.realtime.PriorityScheduler instance()
```

Returns

A reference to the distinguished instance `PriorityScheduler`.

getMaxPriority

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

Signature

```
public
int getMaxPriority()
```

Returns

The value of the maximum priority.

getMaxPriority(Thread)

Gets the maximum priority for the given thread. When the given thread is a realtime thread that is scheduled by an instance of `PriorityScheduler`, then the maximum priority for that scheduler is returned. When the given thread is a Java thread then the maximum priority of its thread group is returned. Otherwise an exception is thrown.

Signature

```
public static
int getMaxPriority(Thread thread)
```

Parameters

thread An instance of `Thread`. When `null`, the maximum priority of this scheduler is returned.

Throws

IllegalArgumentException when `thread` is a realtime thread that is not scheduled by an instance of `PriorityScheduler`.

Returns

The maximum priority for `thread`

getMinPriority

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

Signature

```
public
int getMinPriority()
```

Returns

The minimum priority used by this scheduler.

getMinPriority(Thread)

Gets the minimum priority for the given thread. When the given thread is a realtime thread that is scheduled by an instance of `PriorityScheduler`, then the minimum priority for that scheduler is returned. When the given thread is a Java thread then `Thread.MIN_PRIORITY` is returned. Otherwise an exception is thrown.

Signature

```
public static
int getMinPriority(Thread thread)
```

Parameters

thread An instance of `Thread`. When `null`, the minimum priority of this scheduler is returned.

Throws

IllegalArgumentException when `thread` is a realtime thread that is not scheduled by an instance of `PriorityScheduler`.

Returns

The minimum priority for `thread`

getNormPriority

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

Signature

```
public
int getNormPriority()
```

Returns

The value of the normal priority.

getNormPriority(Thread)

Gets the "norm" priority for the given thread. When the given thread is a realtime thread that is scheduled by an instance of `PriorityScheduler`, then the norm priority for that scheduler is returned. When the given thread is a Java thread then `Thread.NORM_PRIORITY` is returned. Otherwise an exception is thrown.

Signature

```
public static
int getNormPriority(Thread thread)
```

Parameters

thread An instance of `Thread`. When `null`, the norm priority for this scheduler is returned.

Throws

IllegalArgumentException when `thread` is a realtime thread that is not scheduled by an instance3 of `PriorityScheduler`.

Returns

The norm priority for `thread`

getPolicyName

Gets the policy name of `this`.

Signature

```
public
java.lang.String getPolicyName()
```

Returns

The policy name (Fixed Priority) as a string.

6.4.2.7 ProcessingGroupParameters

This is associated with one or more schedulables for which the system guarantees that the associated objects will not be given more time per period than indicated by `cost`.The motivation for this class is to allow the execution demands of one or more aperiodic schedulables to be bound. However, periodic or sporadic schedulables can also be associated with a processing group.

Processing groups have an associated affinity set that must contain only a single processor. The default affinity set is given by `Affinity.getGroupDefaultAffinity()`.

For all schedulables with a reference to an instance of `ProcessingGroupParameters p` no more than `p.cost` will be allocated to the execution of these schedulables on the processor associated with its processing group in each interval of time given by `p.period` after the time indicated by `p.start`. No execution of the schedulables will be allowed on any processor other than this processor. When there is no intersection between the a schedulable objects affinity set and its processing group's affinity set, then the schedulable execution is constrained by the default processing group's affinit set.

Logically a virtual server is associated with each instance of `ProcessingGroupParameters`. This server has a start time, a period, a cost (budget) and a deadline. The server can only logically execute when (a) it has not consumed more execution time in its current release than the cost (budget) parameter, (b) one of its associated schedulables is executable and is the most eligible of the executable schedulables. When the server is logically executable, the associated schedulable is executed. When the cost has been consumed, any `overrunHandler` is released, and the server is not eligible for logical execution until its next period is due. At this point, its allocated cost (budget) is replenished. When the server is logically executing when its deadline expires, any associated `missHandler` is released. The deadline and cost parameters of all the associated schedulable objects have the same impact as they would if the objects were not bound to a processing group.

Processing group parameters use `HighResolutionTime`[88] values for cost, deadline, period and start time. Since those times are expressed as a `HighResolutionTime`[89], the values use accurate timers with nanosecond granularity. The actual resolution available and even the quantity it measures depends on the clock associated with each time value.

When a reference to a `ProcessingGroupParameters` object is given as a parameter to a schedulable's constructor or passed as an argument to one of the schedulable's setter methods, the `ProcessingGroupParameters` object becomes the processing group parameters object bound to that schedulable object. Changes to the values in the `ProcessingGroupParameters` object affect that schedulable object. When bound to more than one schedulable then changes to the values in the `ProcessingGroupParameters` object affect *all* of the associated objects. Note that this is a one-to-many relationship and *not* a many-to-many.

The implementation must use modified copy semantics for each `HighResolutionTime`[90] parameter value. The value of each time object should be treated as if it were copied at the time it is passed to the parameter object, but the object reference must also be retained. Only changes to a `ProcessingGroupParameters` object caused by methods on that object are immediately visible to the scheduler. For instance, invoking `setPeriod()` on a `ProcessingGroupParameters` object will make the change, then notify that the scheduler that the parameter object has changed. At that point the scheduler's view of the processing group parameters object is updated. Invoking a method on the `RelativeTime` object that is the period for this object may change the period but it does not pass the change to the scheduler at that time. That new value for period must not change the behavior of the SOs that use the parameter object until a setter method on the `ProcessingGroupParameters` object is invoked, or the parameter object is used in `setProcessingGroupParameters()` or a constructor for an SO.

The implementation may use copy semantics for each `HighResolutionTime` parameter value. For instance the value returned by `getCost()` must be `equal` to the value passed in by `setCost`, but it need not be the same object.

The following table gives the default parameter values for the constructors.

[88]Section 9.4.1.2
[89]Section 9.4.1.2
[90]Section 9.4.1.2

Attribute	Default Value
start	new RelativeTime(0,0)
period	No default. A value must be supplied
cost	No default. A value must be supplied
deadline	new RelativeTime(period)
overrunHandler	None
missHandler	None

Caution: This class is explicitly unsafe in multithreaded situations when it is being changed. No synchronization is done. It is assumed that users of this class who are mutating instances will be doing their own synchronization at a higher level.

Caution: The `cost` parameter time should be considered to be measured against the target platform.

Inheritance

java.lang.Object
javax.realtime.ProcessingGroupParameters

Interfaces

`Cloneable`
`Serializable`

6.4.2.7.1 Constructors

ProcessingGroupParameters(javax.realtime.HighResolutionTime<T>, RelativeTime, RelativeTime, RelativeTime, AsyncEventHandler, AsyncEventHandler)

Create a `ProcessingGroupParameters` object.

Signature

```
public
  ProcessingGroupParameters(javax.realtime.HighResolutionTime<T> start, Relativ
```

Parameters

start Time at which the first period begins. When a `RelativeTime`, this time is relative to the creation of this. When an `AbsoluteTime`, then the first release of the logical server is at the start time (or immediately when the absolute time is in the past). When `null`, the default value is a new instance of `RelativeTime(0,0)`.
period The period is the interval between successive replenishment of the logical server's associated cost budget. There is no default value. When `period` is `null` an exception is thrown.
cost Processing time per period. The budget CPU time that the logical server can consume each period. When `null`, an exception is thrown.
deadline The latest permissible completion time measured from the start of the current period. Changing the deadline might not take effect after the expiration of the current deadline. Specifying a deadline less than the period constrains execution of all the members of the group to the beginning of each period. When `null`, the default value is new instance of `RelativeTime(period)`.
overrunHandler This handler is invoked when any schedulable object member of this processing group attempts to use processor time beyond the group's budget. When `null`, no application async event handler is fired on the overrun condition.
missHandler This handler is invoked when the logical server is still executing after the deadline has passed. When `null`, no application async event handler is fired on the deadline miss condition.

Throws

IllegalArgumentException when the `period` is `null` or its time value is not greater than zero, when `cost` is `null`, or when the time value of `cost` is less than zero, when `start` is an instance of `RelativeTime` and its value is negative, or when the time value of `deadline` is not greater than zero and less than or equal to the `period`. When the implementation does not support processing group deadline less than period, `deadline` less than `period` will cause `IllegalArgumentException` to be thrown.
IllegalAssignmentError when `start`, `period`, `cost`, `deadline`, `overrunHandler` or `missHandler` cannot be stored in `this`.

6.4.2.7.2 Methods

---

clone

Return a clone of `this`. This method should behave effectively as when it constructed a new object with clones of the high-resolution time values of `this`.

- The new object is in the current allocation context.
- `clone` does not copy any associations from `this` and it does not implicitly bind the new object to a SO.
- The new object has clones of all high-resolution time values (deep copy).
- References to event handlers are copied (shallow copy.)

Available since RTSJ 1.0.1

Signature

```
public
java.lang.Object clone()
```

getCost

Gets the value of `cost`.

Signature

```
public
javax.realtime.RelativeTime getCost()
```

Returns

a reference to the value of `cost`.

getCostOverrunHandler

Gets the cost overrun handler.

Signature

```
public
javax.realtime.AsyncEventHandler getCostOverrunHandler()
```

Returns

A reference to an instance of `AsyncEventHandler`[91] that is cost overrun handler of `this`.

[91] Section 8.4.3.5

getDeadline

Gets the value of `deadline`.

Signature

```
public
javax.realtime.RelativeTime getDeadline()
```

Returns

A reference to an instance of `RelativeTime`[92] that is the deadline of `this`.

getDeadlineMissHandler

Gets the deadline miss handler.

Signature

```
public
javax.realtime.AsyncEventHandler getDeadlineMissHandler()
```

Returns

A reference to an instance of `AsyncEventHandler`[93]that is deadline miss handler of `this`.

getPeriod

Gets the value of `period`.

Signature

```
public
javax.realtime.RelativeTime getPeriod()
```

Returns

A reference to an instance of `RelativeTime`[94] that represents the value of `period`.

[92]Section 9.4.1.3
[93]Section 8.4.3.5
[94]Section 9.4.1.3

getStart

Gets the value of `start`. This is the value that was specified in the constructor or by `setStart()`, not the actual absolute time the corresponding to the start of the processing group.

Signature

```
public
javax.realtime.HighResolutionTime<T> getStart()
```

Returns

A reference to an instance of `HighResolutionTime`[95] that represents the value of `start`.

setCost(RelativeTime)

Sets the value of `cost`.

Signature

```
public
void setCost(RelativeTime cost)
```

Parameters

cost The new value for `cost`. When `null`, an exception is thrown.

Throws

IllegalArgumentException when `cost` is `null` or its time value is less than zero.

IllegalAssignmentError when `cost` cannot be stored in `this`.

setCostOverrunHandler(AsyncEventHandler)

Sets the cost overrun handler.

Signature

```
public
void setCostOverrunHandler(AsyncEventHandler handler)
```

Parameters

[95]Section 9.4.1.2

handler This handler is invoked when the `run()` method of and of the the schedulables attempt to execute for more than `cost` time units in any period. When `null`, no handler is attached, and any previous handler is removed.

Throws

IllegalAssignmentError when `handler` cannot be stored in `this`.

setDeadline(RelativeTime)

Sets the value of `deadline`.

Signature

```
public
void setDeadline(RelativeTime deadline)
```

Parameters

deadline The new value for `deadline`. When `null`, the default value is new instance of `RelativeTime(period)`.

Throws

IllegalArgumentException when `deadline` has a value less than zero or greater than the period. Unless the implementation supports deadline less than period in processing groups, `IllegalArgumentException` is also when `deadline` is less than the period.

IllegalAssignmentError when `deadline` cannot be stored in `this`.

setDeadlineMissHandler(AsyncEventHandler)

Sets the deadline miss handler.

Signature

```
public
void setDeadlineMissHandler(AsyncEventHandler handler)
```

Parameters

handler This handler is invoked when the `run()` method of any of the schedulables still expect to execute after the deadline has passed. When `null`, no handler is attached, and any previous handler is removed.

Throws

IllegalAssignmentError when `handler` cannot be stored in `this`.

setPeriod(RelativeTime)

Sets the value of `period`.

Signature

```
public
void setPeriod(RelativeTime period)
```

Parameters

period The new value for `period`. There is no default value. When `period` is `null` an exception is thrown.

Throws

IllegalArgumentException when `period` is `null`, or its time value is not greater than zero. When the implementation does not support processing group deadline less than period, and `period` is not equal to the current value of the processing group's deadline, the deadline is set to a clone of `period` created in the same memory area as `period`.

IllegalAssignmentError when `period` cannot be stored in `this`.

setStart(javax.realtime.HighResolutionTime<T>)

Sets the value of `start`. When the processing group is already started this method alters the value of this object's start time property, but has no other effect.

Signature

```
public
void setStart(javax.realtime.HighResolutionTime<T> start)
```

Parameters

start The new value for `start`. When `null`, the default value is a new instance of `RelativeTime(0,0)`.

Throws

IllegalAssignmentError when `start` cannot be stored in `this`.

IllegalArgumentException when `start` is a relative time value and less than zero.

6.4.2.8 ReleaseParameters

The top-level class for release characteristics of schedulable objects. When a reference to a ReleaseParameters object is given as a parameter to a constructor of a schedulable, the ReleaseParameters object becomes bound to the object being created. Changes to the values in the ReleaseParameters object affect the constructed object. When given to more than one constructor, then changes to the values in the ReleaseParameters object affect *all* of the associated objects. Note that this is a one-to-many relationship and *not* a many-to-many. Only changes to an ReleaseParameters object caused by methods on that object cause the change to propagate to all schedulables using the object. For instance, invoking setDeadline on a ReleaseParameters instance will make the change, and then notify that the scheduler that the object has been changed. At that point the object is reconsidered for every SO that uses it. Invoking a method on the RelativeTime object that is the deadline for this object may change the time value but it does not pass the new time value to the scheduler at that time. Even though the changed time value is referenced by ReleaseParameters objects, it will not change the behavior of the SOs that use the parameter object until a setter method on the ReleaseParameters object is invoked, or the parameter object is used in setReleaseParameters() or a constructor for a schedulable.

Release parameters use HighResolutionTime[96] values for cost, and deadline. Since the times are expressed as a HighResolutionTime[97] values, these values use accurate timers with nanosecond granularity. The actual resolution available and even the quantity the timers measure depend on the clock associated with each time value.

The implementation must use modified copy semantics for each HighResolutionTime[98] parameter value. The value of each time object should be treated as when it were copied at the time it is passed to the parameter object, but the object reference must also be retained. For instance, the value returned by getCost() must be the same object passed in by setCost(), but any changes made to the time value of the cost must not take effect in the associated ReleaseParameters instance unless they are passed to the parameter object again, e.g. with a new invocation of setCost.

The following table gives the default parameter values for the constructors.

[96]Section 9.4.1.2
[97]Section 9.4.1.2
[98]Section 9.4.1.2

Attribute	Default Value
cost	new RelativeTime(0,0)
deadline	no default
overrunHandler	None
missHandler	None

Caution: This class is explicitly unsafe for multithreading when being changed. Code that mutates instances of this class should synchronize at a higher level.

Inheritance

java.lang.Object
  javax.realtime.ReleaseParameters

Interfaces

`Cloneable`
`Serializable`

6.4.2.8.1 Constructors

ReleaseParameters

Create a new instance of `ReleaseParameters`. This constructor creates a default `ReleaseParameters` object, i.e., it is equivalent to `ReleaseParameters(null, null, null, null)`.

Signature

```
protected
  ReleaseParameters()
```

ReleaseParameters(RelativeTime, AsyncEventHandler)

Create a new instance of `ReleaseParameters` with the given parameter values. This constructor is equivalent to `ReleaseParameters(null, deadline, null, missHandler)`.
Available since RTSJ 2.0

Signature

```
protected
  ReleaseParameters(RelativeTime deadline, AsyncEventHandler missHandler)
```

ReleaseParameters(RelativeTime, RelativeTime, AsyncEventHandler, AsyncEventHandler)

Create a new instance of `ReleaseParameters` with the given parameter values.

Signature

```
protected
  ReleaseParameters(RelativeTime cost, RelativeTime deadline, AsyncEventHandler over
```

Parameters

cost - Processing time units per release. On implementations which can measure the amount of time a schedulable object is executed, When null, the default value is a new instance of RelativeTime(0,0).
deadline - The latest permissible completion time measured from the release time of the associated invocation of the schedulable. There is no default for deadline in this class. The default must be determined by the subclasses.
overrunHandler - This handler is invoked when an invocation of the schedulable exceeds cost. In the minimum implementation overrunHandler is ignored. When null, no application event handler is executed on cost overrun.
missHandler - This handler is invoked when the run() method of the schedulable is still executing after the deadline has passed. When null, no application event handler is executed on the miss deadline condition.

Throws

java.lang.IllegalArgumentException - when the time value of cost is less than zero, or the time value of deadline is less than or equal to zero or the clock associated with the `cost` parameters is not the real-time clock.
IllegalAssignmentError - when cost, deadline, overrunHandler, or missHandler cannot be stored in this.

6.4.2.8.2 Methods

---

clone

Return a clone of `this`. This method should behave effectively as when it constructed a new object with clones of the high-resolution time values of `this`.

- The new object is in the current allocation context.
- `clone` does not copy any associations from `this` and it does not implicitly bind the new object to a SO.
- The new object has clones of all high-resolution time values (deep copy).
- References to event handlers are copied (shallow copy.)

Available since RTSJ 1.0.1

Signature

```
public
java.lang.Object clone()
```

getCost

Gets a reference to the `cost`.

Signature

```
public
javax.realtime.RelativeTime getCost()
```

Returns

A reference to `cost`.

getCostOverrunHandler

Gets a reference to the cost overrun handler.

Signature

```
public
javax.realtime.AsyncEventHandler getCostOverrunHandler()
```

Returns

A reference to the associated cost overrun handler.

getDeadline

Gets a reference to the `deadline`.

Signature

```
public
javax.realtime.RelativeTime getDeadline()
```

Returns

A reference to `deadline`.

getDeadlineMissHandler

Gets a reference to the deadline miss handler.

Signature

```
public
javax.realtime.AsyncEventHandler getDeadlineMissHandler()
```

Returns

A reference to the deadline miss handler.

setCost(RelativeTime)

Sets the cost value.

When this parameter object is associated with any schedulable object (by being passed through the schedulable's constructor or set with a method such as `RealtimeThread.setReleaseParameters(ReleaseParameters)`[99]) the cost of those schedulables is altered as specified by each schedulable's respective scheduler.

Signature

```
public
void setCost(RelativeTime cost)
```

Parameters

[99]Section 5.3.2.2.2

cost Processing time units per release. On implementations which can measure the amount of time a schedulable is executed, this value is the maximum amount of time a schedulable receives per release. On implementations which cannot measure execution time, it is not possible to determine when any particular object exceeds cost. When `null`, the default value is a new instance of `RelativeTime(0,0)`.

Throws

IllegalArgumentException when the time value of `cost` is less than zero, or the clock associated with the `cost` parameters is not the real-time clock.

IllegalAssignmentError when `cost` cannot be stored in `this`.

setCostOverrunHandler(AsyncEventHandler)

Sets the cost overrun handler.

When this parameter object is associated with any schedulable object (by being passed through the schedulable's constructor or set with a method such as `RealtimeThread.setReleaseParameters(ReleaseParameters)`[100]) the cost overrun handler of those schedulables is altered as specified by each schedulable's respective scheduler.

Signature

```
public
void setCostOverrunHandler(AsyncEventHandler handler)
```

Parameters

handler This handler is invoked when an invocation of the schedulable attempts to exceed `cost` time units in a release. A `null` value of `handler` signifies that no cost overrun handler should be used.

Throws

IllegalAssignmentError when `handler` cannot be stored in `this`.

setDeadline(RelativeTime)

Sets the deadline value.

When this parameter object is associated with any schedulable object (by being passed through the schedulable's constructor or set with a method such as `RealtimeThread.setReleaseParameters(ReleaseParameters)`[101]) the deadline of those schedulables is altered as specified by each schedulable's respective scheduler.

[100] Section 5.3.2.2.2

[101] Section 5.3.2.2.2

Signature

```
public
void setDeadline(RelativeTime deadline)
```

Parameters

deadline The latest permissible completion time measured from the release time of the associated invocation of the schedulable. The default value of the deadline must be controlled by the classes that extend `ReleaseParameters`.

Throws

IllegalArgumentException when `deadline` is `null`, the time value of `deadline` is less than or equal to zero, or when the new value of this deadline is incompatible with the scheduler for any associated schedulable.

IllegalAssignmentError when `deadline` cannot be stored in `this`.

setDeadlineMissHandler(AsyncEventHandler)

Sets the deadline miss handler.

When this parameter object is associated with any schedulable object (by being passed through the schedulable's constructor or set with a method such as `RealtimeThread.setReleaseParameters(ReleaseParameters)`[102]) the deadline miss handler of those schedulables is altered as specified by each schedulable's respective scheduler.

Signature

```
public
void setDeadlineMissHandler(AsyncEventHandler handler)
```

Parameters

handler This handler is invoked when any release of the schedulable fails to complete before the deadline passes. A `null` value of `handler` signifies that no deadline miss handler should be used.

Throws

IllegalAssignmentError when `handler` cannot be stored in `this`.

6.4.2.9 Scheduler

An instance of `Scheduler` manages the execution of schedulables.

[102]Section 5.3.2.2.2

Subclasses of `Scheduler` are used for alternative scheduling policies and should define an `instance()` class method to return the default instance of the subclass. The name of the subclass should be descriptive of the policy, allowing applications to deduce the policy available for the scheduler obtained via `Scheduler.getDefaultScheduler`[103] (e.g., `EDFScheduler`).

Inheritance

java.lang.Object
  javax.realtime.Scheduler

6.4.2.9.1 Constructors

Scheduler

Create an instance of `Scheduler`.

Signature

```
protected
  Scheduler()
```

6.4.2.9.2 Methods

---

getDefaultScheduler

Gets a reference to the default scheduler.

Signature

```
public static
javax.realtime.Scheduler getDefaultScheduler()
```

Returns

A reference to the default scheduler.

[103]Section 6.4.2.9.2

setDefaultScheduler(Scheduler)

Sets the default scheduler. This is the scheduler given to instances of schedulables when they are constructed by a Java thread. The default scheduler is set to the required `PriorityScheduler`[104] at startup.

Signature

```
public static
void setDefaultScheduler(Scheduler scheduler)
```

Parameters

scheduler The `Scheduler` that becomes the default scheduler assigned to new schedulables created by Java threads. When `null` nothing happens.

Throws

SecurityException when the caller is not permitted to set the default scheduler.

getPolicyName

Gets a string representing the policy of `this`. The string value need not be interned, but it must be created in a memory area that does not cause an illegal assignment error when stored in the current allocation context and does not cause a `MemoryAccessError`[105] when accessed.

Signature

```
public abstract
java.lang.String getPolicyName()
```

Returns

A **name** object which is the name of the scheduling policy used by `this`.

inSchedulableExecutionContext

Determine whether the current calling context is a `Schedulable`[106]: `Realtime-Thread`[107] or `AbstractAsyncEventHandler`[108].

[104] Section 6.4.2.6
[105] Section 14.2.3.4
[106] Section 6.4.1.2
[107] Section 5.3.2.2
[108] Section 8.4.3.2

Available since RTSJ 2.0

Signature

`public static`
`boolean inSchedulableExecutionContext()`

Returns

`true` when yes and `false` otherwise.

getCurrentSchedulable

Get the current execution context when called from a `Schedulable`[109] execution context.

Available since RTSJ 2.0

Signature

`public static`
`javax.realtime.Schedulable getCurrentSchedulable()`

Throws

ClassCastException when the caller is not a `Schedulable`[110]

Returns

the current `Schedulable`[111].

6.4.2.10 SchedulingParameters

Subclasses of `SchedulingParameters` (`PriorityParameters`[112], `ImportanceParameters`[113], and any others defined for particular schedulers) provide the parameters to be used by the `Scheduler`[114]. Changes to the values in a parameters object affects the scheduling behavior of all the `Schedulable`[115] objects to which it is bound.

[109] Section 6.4.1.2
[110] Section 6.4.1.2
[111] Section 6.4.1.2
[112] Section 6.4.2.5
[113] Section 6.4.2.3
[114] Section 6.4.2.9
[115] Section 6.4.1.2

Caution: This class is explicitly unsafe for multithreading when being changed. Code that mutates instances of this class should synchronize at a higher level.

Inheritance

java.lang.Object
  javax.realtime.SchedulingParameters

Interfaces

`Cloneable`
`Serializable`

6.4.2.10.1 Constructors

SchedulingParameters

Create a new instance of `SchedulingParameters`.
Available since RTSJ 1.0.1

Signature

```
protected
  SchedulingParameters()
```

6.4.2.10.2 Methods

---

clone

Return a clone of `this`. This method should behave effectively as if it constructed a new object with clones of the high-resolution time values of `this`.

- The new object is in the current allocation context.
- `clone` does not copy any associations from `this` and it does not implicitly bind the new object to a SO.
- The new object has clones of all high-resolution time values (deep copy).
- References to event handlers are copied (shallow copy.)

Available since RTSJ 1.0.1

Signature

```
public
java.lang.Object clone()
```

6.4.2.11 SporadicParameters

A notice to the scheduler that the associated schedulable's run method will be released aperiodically but with a minimum time between releases.

When a reference to a `SporadicParameters` object is given as a parameter to a schedulable's constructor or passed as an argument to one of the schedulable's setter methods, the `SporadicParameters` object becomes the release parameters object bound to that schedulable. Changes to the values in the `SporadicParameters` object affect that schedulable object. When bound to more than one schedulable then changes to the values in the `SporadicParameters` object affect *all* of the associated objects. Note that this is a one-to-many relationship and *not* a many-to-many.

The implementation must use modified copy semantics for each `HighResolutionTime`[116] parameter value. The value of each time object should be treated as when it were copied at the time it is passed to the parameter object, but the object reference must also be retained. Only changes to a `SporadicParameters` object caused by methods on that object cause the change to propagate to all schedulables using the parameter object. For instance, calling `setCost` on a `SporadicParameters` object will make the change, then notify that the scheduler that the parameter object has changed. At that point the object is reconsidered for every SO that uses it. Invoking a method on the `RelativeTime` object that is the cost for this object may change the cost but it does not pass the change to the scheduler at that time. That change must not change the behavior of the SOs that use the parameter object until a setter method on the `SporadicParameters` object is invoked, or the parameter object is used in `setReleaseParameters()` or a constructor for an SO.

The following table gives the default parameter values for the constructors.

[116]Section 9.4.1.2

Attribute	Value
minInterarrival time	No default. A value must be supplied
cost	new RelativeTime(0,0)
deadline	new RelativeTime(mit)
overrunHandler	None
missHandler	None
MIT violation policy	SAVE
Arrival queue overflow policy	SAVE
Initial arrival queue length	0

Caution: This class is explicitly unsafe for multithreading when being changed. Code that mutates instances of this class should synchronize at a higher level.

Inheritance

java.lang.Object
  javax.realtime.ReleaseParameters
    javax.realtime.AperiodicParameters
      javax.realtime.SporadicParameters

6.4.2.11.1 Fields

---

mitViolationExcept

`public static final mitViolationExcept`

Represents the "EXCEPT" policy for dealing with minimum interarrival time violations. Under this policy, when an arrival time for any instance of `Schedulable`[117] which has `this` as its instance of `ReleaseParameters`[118] occurs at a time less then the minimum interarrival time defined here then the `fire()` method shall throw `MITViolationException`[119]. Any other associated semantics are governed by the schedulers for the schedulables using these sporadic parameters. When the arrival time is a result of a happening to which the instance of `AsyncEventHandler`[120] is bound then the arrival time is ignored.

[117] Section 6.4.1.2
[118] Section 6.4.2.8
[119] Section 14.2.2.8
[120] Section 8.4.3.5

mitViolationIgnore

`public static final mitViolationIgnore`

Represents the "IGNORE" policy for dealing with minimum interarrival time violations. Under this policy, when an arrival time for any instance of `Schedulable`[121] which has `this` as its instance of `ReleaseParameters`[122] occurs at a time less then the minimum interarrival time defined here then the new arrival time is ignored. Any other associated semantics are governed by the schedulers for the schedulables using these sporadic parameters.

mitViolationSave

`public static final mitViolationSave`

Represents the "SAVE" policy for dealing with minimum interarrival time violations. Under this policy the arrival time for any instance of `Schedulable`[123] which has `this` as its instance of `ReleaseParameters`[124] is not compared to the specified minimum interarrival time. Any other associated semantics are governed by the schedulers for the schedulable objects using these sporadic parameters.

mitViolationReplace

`public static final mitViolationReplace`

Represents the "REPLACE" policy for dealing with minimum interarrival time violations. Under this policy when an arrival time for any instance of `Schedulable`[125] which has `this` as its instance of `ReleaseParameters`[126] occurs at a time less then the minimum interarrival time defined here then the information for this arrival replaces a previous arrival. Any other associated semantics are governed by the schedulers for the schedulables using these sporadic parameters.

6.4.2.11.2 Constructors

---

[121] Section 6.4.1.2
[122] Section 6.4.2.8
[123] Section 6.4.1.2
[124] Section 6.4.2.8
[125] Section 6.4.1.2
[126] Section 6.4.2.8

SporadicParameters(RelativeTime)

Create a `SporadicParameters` object. This constructor is equivalent to: `SporadicParameters(minInterarrival, null, null, null, null)`
Available since RTSJ 1.0.1

Signature

```
public
  SporadicParameters(RelativeTime minInterarrival)
```

SporadicParameters(RelativeTime, RelativeTime)

Create a `SporadicParameters` object. This constructor is equivalent to: `SporadicParameters(minInterarrival, null, deadline, null, null)`
Available since RTSJ 2.0

Signature

```
public
  SporadicParameters(RelativeTime minInterarrival, RelativeTime deadline)
```

SporadicParameters(RelativeTime, RelativeTime, AsyncEventHandler)

Create a `SporadicParameters` object. This constructor is equivalent to: `SporadicParameters(minInterarrival, null, deadline, null, missHandler)`
Available since RTSJ 2.0

Signature

```
public
  SporadicParameters(RelativeTime minInterarrival, RelativeTime deadline, AsyncEvent
```

SporadicParameters(RelativeTime, RelativeTime, RelativeTime, AsyncEventHandler, AsyncEventHandler)

Create a `SporadicParameters` object.
Available since RTSJ 2.0

Signature

```
public
  SporadicParameters(RelativeTime minInterarrival, RelativeTime cost, RelativeT
```

Parameters

minInterarrival The release times of the schedulable will occur no closer than this interval. This time object is treated as if it were copied. Changes to `minInterarrival` will not effect the `SporadicParameters` object. There is no default value. When `minInterarrival` is `null` an illegal argument exception is thrown.
cost Processing time per release. On implementations which can measure the amount of time a schedulable is executed, this value is the maximum amount of time a schedulable receives per release. When `null`, the default value is a new instance of `RelativeTime(0,0)`.
deadline The latest permissible completion time measured from the release time of the associated invocation of the schedulable. When `null`, the default value is a new instance of `RelativeTime(mit)`.
overrunHandler This handler is invoked when an invocation of the schedulable exceeds cost. Not required for minimum implementation. When `null` no overrun handler will be used.
missHandler This handler is invoked when the `run()` method of the schedulable is still executing after the deadline has passed. When `null`, no deadline miss handler will be used.

Throws

IllegalArgumentException when `minInterarrival` is `null` or its time value is not greater than zero, or the time value of `cost` is less than zero, or the time value of `deadline` is not greater than zero, or when the clocks associated with `deadline` and `minInterarrival` parameters are not identical.
IllegalAssignmentError when `minInterarrival`, `cost`, `deadline`, `overrunHandler` or `missHandler` cannot be stored in `this`.

6.4.2.11.3 Methods

setMitViolationBehavior(String)

Sets the behavior of the arrival time queue in the case where the new arrival time is closer to the previous arrival time than the minimum interarrival time given in this.

Values of `behavior` are compared using reference equality (==) not value equality (`equals()`).

Signature

```
public
void setMitViolationBehavior(String behavior)
```

Parameters

behavior A string representing the behavior.

Throws

IllegalArgumentException when `behavior` is not one of the `final` MIT violation behavior values defined in this class.

getMitViolationBehavior

Gets the arrival time queue behavior in the event of a minimum interarrival time violation.

Signature

```
public
java.lang.String getMitViolationBehavior()
```

Returns

The minimum interarrival time violation behavior as a string.

getMinimumInterarrival

Gets the minimum interarrival time.

Signature

```
public
javax.realtime.RelativeTime getMinimumInterarrival()
```

Returns

The minimum interarrival time.

setMinimumInterarrival(RelativeTime)

Set the minimum interarrival time.

Signature

```
public
void setMinimumInterarrival(RelativeTime minimum)
```

Parameters

minimum The release times of the schedulable will occur no closer than this interval.

Throws

IllegalArgumentException when `minimum` is `null` or its time value is not greater than zero.
IllegalAssignmentError when `minimum` cannot be stored in `this`.

6.5 Rationale

As specified, the required semantics of this section establish a scheduling policy that is very similar to the scheduling policies found on the vast majority of realtime operating systems and kernels in commercial use today. The semantics for the base scheduler accommodate existing practice, which is a stated goal of the effort.

There is an important division between priority schedulers that force periodic context switching between tasks at the same priority, and those that do not cause these context switches. By not specifying time slicing[1] behavior this specification calls for the latter type of priority scheduler. In POSIX terms, `SCHED_FIFO` meets the `RTSJ` requirements for the base scheduler, but `SCHED_RR` does not meet those requirements.

Although a system may not implement the first release (start) of a schedulable as unblocking that schedulable, under the base scheduler those semantics apply; i.e., the schedulable is added to the tail of the queue for its active priority.

Some research shows that, given a set of reasonable common assumptions, 32 distinct priority levels are a reasonable choice for close-to-optimal scheduling efficiency when using the rate-monotonic priority assignment algorithm on a single processor system (256 priority levels provide better efficiency). This specification requires at least 28 distinct priority levels as a compromise noting that implementations of this specification will exist on systems with logic executing outside of the Java Virtual Machine and may need priorities above, below, or both for system activities.

The default behavior for implementations that support cost monitoring and enforcement is that a schedulable receives no more than `cost` units of CPU time during each release. The programmer must explicitly change the cost attribute to override the scheduler. The `RTSJ` allows schedulables to self suspend during a release, in addition to that which might be necessary to acquire a lock. These self suspensions must be time bounded. Any self suspension which is not time bounded may undermine the cost enforcement model specified in this document, as it may result in a schedulable suspending beyond its next release event. This can result in more time being allocated than any associated schedulability analysis might assume. See Dos Santos and Wellings for a full discussion on the problem [4].

Cost enforcement may be deferred while the overrun schedulable holds locks that are out of application control, such as locks used to protect garbage collection. Applications should include the resulting jitter in any analysis that depends on cost enforcement.

When a schedulable is enforced because of cost overrun in a processing group the enforced priority is used for scheduling instead of the schedulable's base priority. The enforced priority's application is limited. The enforced priority is not returned as the schedulable's priority from methods such as getPriority(), and the semantics of the active priority continue to operate when a schedulable is enforced.

6.5.1 Multiprocessor Support

The support that the `RTSJ` provides for multiprocessor systems is primarily constrained by the support it can expect from the underlying operating system. The following have had the most impact on the level of support that has been specified.

- The notion of processor *affinity* is common across operating systems and has become the accepted way to specify the constraints on which processor a thread can execute. In some sense, processor affinities can be viewed as additional release or scheduling parameters. However, to add them to the parameter classes requires the support to be distributed throughout the specification with a proliferation of new constructor methods. To avoid this, support is grouped together within the `Affinity` class. The class also provides the addition of processor affinity support to Java threads without modifying the thread object's visible API.
- The range of processors on which global scheduling is possible is dictated by the operating system. For SMP architectures, global scheduling across all the processors in the system is typically supported. However, an application and an operator can constrain threads and processes to execute only within a

subset of the processors. As the number of processors increase, the scalability of global scheduling is called into question. Hence, for NUMA architectures some partitioning of the processors is likely to performed by the OS. Hence, global scheduling across all processors will not be possible in these systems. For these reasons, the `RTSJ` supports an array of *predefined* affinities. These are implementation-defined. They can be used either to reflect the scheduling arrangement of the underlying OS or they can be used by the system designer to impose defaults for, say, Java threads, non-heap realtime schedulables etc. A program is only allowed to dynamically create new affinities with cardinality of one. This restriction reflects the concern that not all operating systems will support multiprocessor affinities.

- Many OSs give system operators command-level dynamic control over the set of processors allocated to a processes. Consequently, the realtime JVM has no control over whether processors are dynamically added or removed from its OS process. Predictability is a prime concern of the `RTSJ`. Clearly, dynamic changes to the allocated processors will have a dramatic, and possibly catastrophic, effect on the ability of the program to meet timing requirements. Hence, the `RTSJ` assumes that the processor set allocated to the `RTSJ` process does not change during its execution. If a system is capable of such manipulations, it should not exercise in on `RTSJ` processes.
- The reason we decided not to add affinities to scheduling parameters is that ASEH do not have a single server thread, hence forcing a particular affinity would complicate the implementation.

6.5.2 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 $T0$, where $T0 >= T$. The upper bound of δ is the value returned from calling the getResolution method of the associated clock. It is for this reason that the implementation of release times for periodic activities must use absolute rather than relative time values, in order to avoid the drift accumulating.

The second contribution to release jitter is also related to the clock/timer. It is the duration of interval between $T0$ 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 disabled). A compliant implementation of SCJ should document the maximum value of δ for the realtime clock.

6.5.3 Deadline Miss Detection

Although `RTSJ` 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. 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 schedulables.

A related limitation is that a deadline can be missed but not detected. This can occur when the deadline has been set at a smaller granularity than the detecting timer. Consider an absolute deadline of D. Suppose that the next absolute time that the timer can recognize is $D + \delta$. If the associate thread finishes after D but before $D + \delta$, it will have missed its deadline, but this miss will have been undetected.

A third limitation is due to the inherent race condition that is present when checking for deadline misses. A deadline miss is defined to occur if a schedulable 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 or a call to waitForNextPeriod etc. When this occurs, 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 the completion event 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.

Chapter 7

Synchronization

(Updated by Andy 8 Feb, 2012)

7.1 Overview

This section describes classes that specifically manage synchronization. These classes:

- Allow the setting of a priority inversion control policy either as the default or for specific objects
- Allow wait-free communication between schedulables (especially instances of `NoHeapRealtimeThread`) and regular Java threads.

This specification strengthens the semantics of Java `synchronized` code by mandating monitor execution eligibility control, commonly referred to as priority inversion control. The `MonitorControl` class is defined as the superclass of all such execution eligibility control algorithms. Its subclasses `PriorityInheritance` (required) and `PriorityCeilingEmulation` (optional) avoid unbounded priority inversions, which would be unacceptable in realtime systems.

The classes in this section establish a framework for priority inversion management that applies to priority-oriented schedulers in general, and a specific set of requirements for the base priority scheduler.

The wait-free queue classes provide safe, concurrent access to data shared between instances of `NoHeapRealtimeThread` and schedulable objects subject to garbage collection delays.

7.2 Semantics

This list establishes the semantics that are applicable across the classes of this section. Semantics that apply to particular classes, constructors, methods, and fields will be found in the class description and the constructor, method, and field detail sections.

- Terminology: If an object `obj` has been assigned (either by default or via an explicit method call) the `MonitorControlPolicy mcp`, then `obj` is said to be *governed by* `mcp`.
- The initial default monitor control policy shall be `PriorityInheritance`. The default policy can be altered by using the `setMonitorControl()` method.
- Notwithstanding the preceding rule, an `RTSJ` implementation may allow the program to establish a different initial default monitor control policy at JVM startup. The program can query the initial default monitor control policy via the method `RealtimeSystem.getInitialMonitorControl`.
- The `PriorityCeilingEmulation` monitor control policy is optional, since it is not widely supported by current RTOSes.
- An implementation that provides any additional `MonitorControl` subclasses must document their effects, particularly with respect to priority inversion control. X
- An object's monitor control policy affects *any* entity that attempts to lock the object; i.e., regular Java threads as well as schedulables.
- When a thread or schedulable enters synchronized code, the target object's monitor control policy must be supported by the thread or schedulable's scheduler; otherwise an `IllegalThreadStateException` is thrown. An implementation that defines a new `MonitorControl` subclass must document which (if any) schedulers do not support this policy.

7.2.1 The Base Priority Scheduler

The following list defines the main terms and establishes the general semantics that apply to threads and schedulables managed by the base priority scheduler when they synchronize on objects governed by monitor control policies defined in this section.

- Each thread or schedulable has a *base priority* and an *active priority.* A thread or schedulable that holds a lock on a PCE-governed object also has a *ceiling priority.*
- The *base priority* for a thread or schedulable `t` is initially the priority that `t` has when it is created. The base priority is updated (immediately) as an effect of invoking any of the following methods:
  - `pparams.setPriority(prio)` if `t` is a schedulable with `pparams` as its `SchedulingParameters`, where `pparams` is an instance of `PriorityParameters`; the new base priority is `prio`
  - `t.setSchedulingParameters(pparams)` if `t` is a schedulable and `pparams` is an instance of `PriorityParameters`; the new base priority is `pparams.getPriority()`

  - `t.setPriority(prio)` if `t` is a schedulable object, the new base priority is `prio`. If it is a Java thread, the new base priority is the lesser of `prio`, and the maximum priority for `t`'s thread group.

- When `t` does not hold any locks, its active priority is the same as its base priority. In such a situation modification of the priority of `t` through an invocation of any of the above priority-setting methods for `t` causes `t` to be placed at the tail of its relevant queue (ready, blocked on a particular object, etc.) at its new priority.
- When `t` holds one or more locks, then `t` has a set of *priority sources*. The *active priority* for `t` at any point in time is the maximum of the priorities associated with all of these sources. The priority sources resulting from the monitor control policies defined in this section, and their associated priorities for a schedulable `t`, are as follows:
  - *Source*: `t` itself *Associated priority*: The base priority for `t`*Note*: This may have been changed (either synchronously or asynchronously) while `t` has been holding its lock(s).
  - *Source*: Each object locked by `t` and governed by a `PriorityCeilingEmulation` policy *Associated priority*: The maximum value `ceil` such that `ceil` is the ceiling for a `PriorityCeilingEmulation` policy governing an object locked by `t`. This value is also referred to as the *ceiling priority* for `t`.
  - *Source*: Each thread or schedulable that is attempting to synchronize on an object locked by `t` and governed by a `PriorityInheritance` policy *Associated priority*: The maximum active priority over all such threads and schedulables *Note*: This rule accounts for recursive priority inheritance.
  - Source: Each thread or schedulable that is attempting to synchronize on an object locked by `t` and governed by a `PriorityCeilingEmulation` policy. *Associated priority*: The maximum active priority over all such threads and schedulables
    Note: This rule, which in effect allows a `PriorityCeilingEmulation` lock to behave like a `PriorityInheritance` lock, helps avoid unbounded priority inversions that could otherwise occur in the presence of nested synchronizations involving a mix of `PriorityCeilingEmulation` and `PriorityInheritance` policies.

- The addition of a priority source for `t` either leaves `t`'s active priority unchanged, or increases it. If `t`'s active priority is unchanged, then `t`'s status in its relevant queue(s) (e.g. blocked waiting for some object) is not affected. If `t`'s active priority is increased, then `t` is placed at the tail of the relevant queue(s) at its new active priority level.

- The removal of a priority source for `t` either leaves `t`'s active priority unchanged, or decreases it. If `t`'s active priority is unchanged, then `t`'s status in its relevant queue(s) (e.g. blocked waiting for some object) is not affected. If `t`'s active priority is decreased and `t` is either ready or running, then `t` must be placed at the head of the ready queue(s) at its new active priority level, if the implementation is supporting `PriorityCeilingEmulation`. If the implementation is not supporting `PriorityCeilingEmulation` then `t` should be placed at the head of the ready queue(s) at its new active priority (Note the "should": this behavior is optional.) If `PriorityCeilingEmulation` is not supported, the implementation must document the queue placement effect. If `t`'s active priority is decreased and `t` is blocked, then `t` is placed in the corresponding queue(s) at its new active priority level. Its position in the queue(s) is implementation defined, but placement at the tail is recommended.

The above rules have the following consequences:

- A thread or schedulable `t`'s priority sources from 4.b are added and removed synchronously; i.e., they are established based on `t`'s entering or leaving synchronized code. However, priority sources from 4.a, 4.c and 4.d may be added and removed asynchronously, as an effect of actions by other threads or schedulables.
- If a thread or schedulable holds only one lock then, when it releases this lock, its active priority is set to its base priority.
- A thread or schedulable's active priority is never less than its base priority.
- When a thread or schedulable blocks at a call of `obj.wait()` it releases the lock on `obj` and hence relinquishes the priority source(s) based on `obj`'s monitor control policy. The thread or schedulable will be queued at a new active priority that reflects the loss of these priority sources.

Since base priorities may be shared (i.e., the same `PriorityParameters` object may be associated with multiple schedulables), a given base priority may be the active priority for some but not all of its associated schedulables. It is a consequence of other rules that, when a thread or schedulable `t` attempts to synchronize on an object `obj` governed by a `PriorityCeilingEmulation` policy with ceiling `ceil`, then `t`'s active priority may exceed `ceil` but `t`'s base priority must not. In contrast, once `t` has successfully synchronized on `obj` then `t`'s base priority may also exceed `obj`'s monitor control policy's ceiling. Note that `t`'s base priority and/or `obj`'s monitor control policy may have been dynamically modified.

7.2.2 Additional Schedulers

The following list establishes the semantics that apply to threads or schedulables managed by a scheduler other than the base priority scheduler when they synchronize on objects with monitor control policies defined in this section.

- An implementation that defines a new `Scheduler` subclass must document which (if any) monitor control policies the new scheduler does not support.
- An implementation must document how, if at all, the semantics of synchronization differ from the rules defined for the default `PriorityInheritance` instance. It must supply documentation for the behavior of the new scheduler with priority inheritance (and, if it is supported, priority ceiling emulation protocol) equivalent to the semantics for the default priority scheduler found in the previous section.

7.3 Package javax.realtime

7.3.1 Classes

7.3.1.1 MonitorControl

Abstract superclass for all monitor control policy objects.

Inheritance

java.lang.Object
javax.realtime.MonitorControl

7.3.1.1.1 Constructors

MonitorControl

Invoked from subclass constructors.

Signature

```
protected
  MonitorControl()
```

7.3.1.1.2 Methods

getMonitorControl(Object)

Gets the monitor control policy of the given instance of `Object`.

Signature

```
public static
javax.realtime.MonitorControl getMonitorControl(Object obj)
```

Parameters

obj The object being queried.

Throws

IllegalArgumentException when `obj` is `null`.

Returns

The monitor control policy of the `obj` parameter.

getMonitorControl

Gets the current default monitor control policy.

Signature

```
public static
javax.realtime.MonitorControl getMonitorControl()
```

Returns

The default monitor control policy object.

setMonitorControl(MonitorControl)

Sets the *default monitor control policy*. This policy does not affect the monitor control policy of any already created object, it will, however, govern any object subsequently constructed, until either:

1. a new "per-object" policy is set for that object. This will alter the monitor control policy for a single object without changing the default policy.
2. a new default policy is set.

Like the per-object method (see `setMonitorControl(Object, MonitorControl)`[1], the setting of the default monitor control policy occurs immediately.

Available since RTSJ 1.0.1 The return type is changed from void to `MonitorControl`.

Signature

```
public static
javax.realtime.MonitorControl setMonitorControl(MonitorControl
policy)
```

Parameters

policy The new monitor control policy. When `null` nothing happens.

Throws

[1]Section 7.3.1.1.2

SecurityException when the caller is not permitted to alter the default monitor control policy.
IllegalArgumentException when `policy` is not in immortal memory.
UnsupportedOperationException when `policy` is not a supported monitor control policy.

Returns

The default `MonitorControl` policy that was replaced.

setMonitorControl(Object, MonitorControl)

Immediately sets `policy` as the monitor control policy for `obj`.

A thread or schedulable that is queued for the lock associated with `obj`, or is in `obj`'s wait set, is not rechecked (e.g., for a `CeilingViolationException`) under `policy`, either as part of the execution of `setMonitorControl` or when it is awakened to (re)acquire the lock.

The thread or schedulable invoking `setMonitorControl` must already hold the lock on `obj`.

Available since RTSJ 1.0.1 The return type is changed from void to `MonitorControl`.

Signature

```
public static
javax.realtime.MonitorControl setMonitorControl(Object obj,
MonitorControl policy)
```

Parameters

obj The object that will be governed by the new policy.
policy The new policy for the object. When `null` nothing will happen.

Throws

IllegalArgumentException Thrown when `obj` is `null` or `policy` is not in immortal memory.
UnsupportedOperationException when `policy` is not a supported monitor control policy.
IllegalMonitorStateException when the caller does not hold a lock on `obj`.

Returns

The current `MonitorControl` policy for `obj`, which will be replaced.

7.3.1.2 PriorityCeilingEmulation

Monitor control class specifying the use of the priority ceiling emulation protocol (also known as the "highest lockers" protocol). Each `PriorityCeilingEmulation` instance is immutable; it has an associated *ceiling*, initialized at construction and queryable but not updatable thereafter.

When a thread or schedulable synchronizes on a target object governed by a `PriorityCeilingEmulation` policy, then the target object becomes a priority source for the thread or schedulable object. When the object is unlocked, it ceases serving as a priority source for the thread or schedulable. The practical effect of this rule is that the thread or schedulable's active priority is boosted to the policy's ceiling when the object is locked, and is reset when the object is unlocked. The value that it is reset to may or may not be the same as the active priority it held when the object was locked; this depends on other factors (e.g. whether the thread or schedulable's base priority was changed in the interim).

The implementation must perform the following checks when a thread or schedulable `t` attempts to synchronize on a target object governed by a `PriorityCeilingEmulation` policy with ceiling `ceil`:

- `t`'s base priority does not exceed `ceil`
- `t`'s ceiling priority (when `t` is holding any other `PriorityCeilingEmulation` locks) does not exceed `ceil`.

Thus for any object `targetObj` that will be governed by priority ceiling emulation, the programmer needs to provide (via `MonitorControl.setMonitorControl(Object, MonitorControl)`[2]) a `PriorityCeilingEmulation` policy whose ceiling is at least as high as the maximum of the following values:

- the highest base priority of any thread or schedulable that could synchronize on `targetObj`
- the maximum ceiling priority value that any thread or schedulable object could have when it attempts to synchronize on `targetObj`.

More formally:

- When a thread or schedulable `t` whose base priority is `p1` attempts to synchronize on an object governed by a `PriorityCeilingEmulation` policy with ceiling `p2`, where `p1>p2`, then a `CeilingViolationException` is thrown in `t`. A `CeilingViolationException` is likewise thrown in `t` when `t` is holding a `PriorityCeilingEmulation` lock and has a ceiling priority exceeding `p2`.

The values of `p1` and `p2` are passed to the constructor for the exception and may be queried by an exception handler.

[2]Section 7.3.1.1.2

A consequence of the above rule is that a thread or schedulable may nest synchronizations on `PriorityCeilingEmulation`-governed objects as long as the ceiling for the inner lock is not less than the ceiling for the outer lock.

The possibility of nested synchronizations on objects governed by a mix of `PriorityInheritance` and `PriorityCeilingEmulation` policies requires one other piece of behavior in order to avoid unbounded priority inversions. When a thread or schedulable holds a `PriorityInheritance` lock, then any `PriorityCeilingEmulation` lock that it either holds or attempts to acquire will exhibit priority inheritance characteristics. This rule is captured above in the definition of priority sources (4.d).

When a thread or schedulable `t` attempts to synchronize on a `PriorityCeilingEmulation`-governed object with ceiling `ceil`, then `ceil` must be within the priority range allowed by `t`'s scheduler; otherwise, an `IllegalThreadStateException` is thrown. Note that this does not prevent a regular Java thread from synchronizing on an object governed by a `PriorityCeilingEmulation` policy with a ceiling higher than 10.

The priority ceiling for an object `obj` can be modified by invoking `MonitorControl.setMonitorControl(obj, newPCE)` where `newPCE`'s ceiling has the desired value.

See also `MonitorControl`[3] `PriorityInheritance`[4], and `CeilingViolationException`[5].

Inheritance

java.lang.Object
  javax.realtime.MonitorControl
    javax.realtime.PriorityCeilingEmulation

7.3.1.2.1 Methods

---

instance(int)

Return a `PriorityCeilingEmulation` object with the specified ceiling. This object is in `ImmortalMemory`. All invocations with the same ceiling value return a reference to the same object.

Available since RTSJ 1.0.1

[3]Section 7.3.1.1
[4]Section 7.3.1.3
[5]Section 14.2.2.2

Signature

```
public static
javax.realtime.PriorityCeilingEmulation instance(int ceiling)
```

Parameters

ceiling Priority ceiling value.

Throws

IllegalArgumentException when `ceiling` is outside of the range of permitted priority values (e.g., less than `PriorityScheduler.instance().getMinPriority()` or greater than `PriorityScheduler.instance().getMaxPriority()` for the base scheduler).

getCeiling

Gets the priority ceiling for this `PriorityCeilingEmulation` object.
Available since RTSJ 1.0.1

Signature

```
public
int getCeiling()
```

Returns

The priority ceiling.

getMaxCeiling

Gets a `PriorityCeilingEmulation` object whose ceiling is `PriorityScheduler.instance().get`
This method returns a reference to a `PriorityCeilingEmulation` object allocated in immortal memory. All invocations of this method return a reference to the same object.
Available since RTSJ 1.0.1

Signature

```
public static
javax.realtime.PriorityCeilingEmulation getMaxCeiling()
```

Returns

A `PriorityCeilingEmulation` object whose ceiling is `PriorityScheduler.instance().getMaxPri`

7.3.1.3 PriorityInheritance

Singleton class specifying use of the priority inheritance protocol. When a thread or schedulable `t1` attempts to enter code that is synchronized on an object `obj` governed by this protocol, and `obj` is currently locked by a lower-priority thread or schedulable `t2`, then

1. When `t1`'s active priority does not exceed the maximum priority allowed by `t2`'s scheduler, then `t1` becomes a priority source for `t2`; `t1` ceases to serve as a priority source for `t2` when either `t2` releases the lock on `obj`, or `t1` ceases attempting to synchronize on `obj` (e.g., when `t1` incurs an ATC).
2. Otherwise (i.e., `t1`'s active priority exceeds the maximum priority allowed by `t2`'s scheduler), an `IllegalThreadStateException` is thrown in `t1`.

Note on the 2nd rule: throwing the exception in `t1`, rather than in `t2`, ensures that the exception is synchronous.

See also `MonitorControl`[6] and `PriorityCeilingEmulation`[7]

Inheritance

java.lang.Object
  javax.realtime.MonitorControl
    javax.realtime.PriorityInheritance

7.3.1.3.1 Methods

instance

Return a reference to the singleton `PriorityInheritance`.
This is the default `MonitorControl` policy in effect at system startup.
The `PriorityInheritance` instance shall be allocated in `ImmortalMemory`.

Signature

```
public static
javax.realtime.PriorityInheritance instance()
```

[6]Section 7.3.1.1
[7]Section 7.3.1.2

7.3.1.4 WaitFreeDequeue

A `WaitFreeDequeue` encapsulates a `WaitFreeWriteQueue` and a `WaitFreeReadQueue`. Each method on a `WaitFreeDequeue` corresponds to an equivalent operation on the underlying `WaitFreeWriteQueue` or `WaitFreeReadQueue`.

Incompatibility with V1.0: Three exceptions previously thrown by the constructor have been deleted from the `throws` clause. These are:

- `java.lang.IllegalAccessException`,
- `java.lang.ClassNotFoundException`, and
- `java.lang.InstantiationException`.

Including these exceptions on the `throws` clause was an error. Their deletion may cause compile-time errors in code using the previous constructor. The repair is to remove the exceptions from the `catch` clause around the constructor invocation.

`WaitFreeDequeue` is one of the classes allowing `NoHeapRealtimeThreads` and regular Java threads to synchronize on an object without the risk of a `NoHeapRealtimeThread` incurring Garbage Collector latency due to priority inversion avoidance management.

Deprecated since RTSJ version as of RTSJ 1.0.1

Inheritance

java.lang.Object
  javax.realtime.WaitFreeDequeue

7.3.1.5 WaitFreeReadQueue

A queue that can be non-blocking for consumers. The `WaitFreeReadQueue` class is intended for single-reader multiple-writer communication, although it may also be used (with care) for multiple readers. A *reader* is generally an instance of `NoHeapRealtimeThread`[8], and the *writers* are generally regular Java threads or heap-using realtime threads or schedulables. Communication is through a bounded buffer of Objects that is managed first-in-first-out. The principal methods for this class are write and read

- The `write` method appends a new element onto the queue. It is synchronized, and blocks when the queue is full. It may be called by more than one writer, in which case, the different callers will write to different elements of the queue.

[8]Section 15.3.2.1

- The `read` method removes the oldest element from the queue. It is not synchronized and does not block; it will return `null` when the queue is empty.Multiple reader threads or schedulables are permitted, but when two or more intend to read from the same `WaitFreeWriteQueue` they will need to arrange explicit synchronization.

For convenience, and to avoid requiring a reader to poll until the queue is non-empty, this class also supports instances that can be accessed by a reader that blocks on queue empty. To obtain this behavior, the reader needs to invoke the `waitForData()` method on a queue that has been constructed with a `notify` parameter set to `true`.

`WaitFreeReadQueue` is one of the classes allowing `NoHeapRealtimeThreads` and regular Java threads to synchronize on an object without the risk of a `NoHeapRealtimeThread` incurring Garbage Collector latency due to priority inversion avoidance management. *Incompatibility with V1.0:* Three exceptions previously thrown by the constructor have been deleted. These are

- `java.lang.IllegalAccessException`,
- `java.lang.ClassNotFoundException`, and
- `java.lang.InstantiationException`.

These exceptions were in error. Their deletion may cause compile-time errors in code using the previous constructor. The repair is to remove the exceptions from the catch clause around the constructor invocation.

Inheritance

java.lang.Object
  javax.realtime.WaitFreeReadQueue

7.3.1.5.1 Constructors

WaitFreeReadQueue(Runnable, Runnable, int, MemoryArea, boolean)

Constructs a queue containing up to `maximum` elements in `memory`. The queue has an unsynchronized and nonblocking `read()` method and a synchronized and blocking `write()` method.

The `writer` and `reader` parameters, when non-null, are checked to insure that they are compatible with the `MemoryArea` specified by `memory` (when non-null.) When `memory` is `null` and both Runnables are non-null, the constructor will select the nearest common scoped parent memory area, or when there is no such scope it will use immortal memory. When all three parameters are `null`, the queue will be allocated in immortal memory.

`reader` and `writer` are not necessarily the only threads or schedulable objects that will access the queue; moreover, there is no check that they actually access the queue at all.

Note: that the wait free queue's internal queue is allocated in `memory`, but the memory area of the wait free queue instance itself is determined by the current allocation context.

Signature

```
public
  WaitFreeReadQueue(Runnable writer, Runnable reader, int maximum, MemoryArea m
```

Parameters

writer An instance of `Runnable` or `null`.
reader An instance of `Runnable` or `null`.
maximum The maximum number of elements in the queue.
memory The `MemoryArea`[9] in which internal elements are allocated.
notify A flag that establishes whether a reader is notified when the queue becomes non-empty.

Throws

IllegalArgumentException when an argument holds an invalid value. The `writer` argument must be `null`, a reference to a `Thread`, or a reference to a schedulable (a `RealtimeThread`, or an `AsyncEventHandler`.) The `reader` argument must be `null`, a reference to a `Thread`, or a reference to a schedulable. The `maximum` argument must be greater than zero.
InaccessibleAreaException when `memory` is a scoped memory that is not on the caller's scope stack.
MemoryScopeException when either `reader` or `writer` is non-null and the `memory` argument is not compatible with `reader` and `writer` with respect to the assignment and access rules for memory areas.

WaitFreeReadQueue(Runnable, Runnable, int, MemoryArea)

Constructs a queue containing up to `maximum` elements in `memory`. The queue has an unsynchronized and nonblocking `read()` method and a synchronized and blocking `write()` method.

[9]Section 11.4.3.6

The `writer` and `reader` parameters, when non-null, are checked to insure that they are compatible with the `MemoryArea` specified by `memory` (when non-null.) When `memory` is `null` and both Runnables are non-null, the constructor will select the nearest common scoped parent memory area, or when there is no such scope it will use immortal memory. When all three parameters are `null`, the queue will be allocated in immortal memory.

`reader` and `writer` are not necessarily the only threads or schedulables that will access the queue; moreover, there is no check that they actually access the queue at all.

Note: that the wait free queue's internal queue is allocated in `memory`, but the memory area of the wait free queue instance itself is determined by the current allocation context.

Signature

```
public
  WaitFreeReadQueue(Runnable writer, Runnable reader, int maximum, MemoryArea memory
```

Parameters

writer An instance of `Runnable` or `null`.

reader An instance of `Runnable` or `null`.

maximum The maximum number of elements in the queue.

memory The `MemoryArea`[10] in which this object and internal elements are allocated.

Throws

IllegalArgumentException when an argument holds an invalid value. The `writer` argument must be `null`, a reference to a `Thread`, or a reference to a schedulable (a `RealtimeThread`, or an `AsyncEventHandler`.) The `reader` argument must be `null`, a reference to a `Thread`, or a reference to a schedulable. The `maximum` argument must be greater than zero.

MemoryScopeException when either `reader` or `writer` is non-null and the `memory` argument is not compatible with `reader` and `writer` with respect to the assignment and access rules for memory areas.

InaccessibleAreaException when `memory` is a scoped memory that is not on the caller's scope stack.

WaitFreeReadQueue(int, MemoryArea, boolean)

[10] Section 11.4.3.6

Constructs a queue containing up to `maximum` elements in `memory`. The queue has an unsynchronized and nonblocking `read()` method and a synchronized and blocking `write()` method.

Note: that the wait free queue's internal queue is allocated in `memory`, but the memory area of the wait free queue instance itself is determined by the current allocation context.

Available since RTSJ 1.0.1

Signature

```
public
  WaitFreeReadQueue(int maximum, MemoryArea memory, boolean notify)
```

Parameters

maximum The maximum number of elements in the queue.

memory The `MemoryArea`[11] in which this object and internal elements are allocated.

notify A flag that establishes whether a reader is notified when the queue becomes non-empty.

Throws

IllegalArgumentException when the `maximum` argument is less than or equal to zero, or `memory` is `null`.

InaccessibleAreaException when `memory` is a scoped memory that is not on the caller's scope stack.

WaitFreeReadQueue(int, boolean)

Constructs a queue containing up to `maximum` elements in immortal memory. The queue has an unsynchronized and nonblocking `read()` method and a synchronized and blocking `write()` method.

Available since RTSJ 1.0.1

Signature

[11] Section 11.4.3.6

```
public
  WaitFreeReadQueue(int maximum, boolean notify)
```

Parameters

maximum The maximum number of elements in the queue.

notify A flag that establishes whether a reader is notified when the queue becomes non-empty.

Throws

IllegalArgumentException when the `maximum` argument is less than or equal to zero.

7.3.1.5.2 Methods

---

clear

Sets `this` to empty.

Note: This method needs to be used with care. Invoking `clear` concurrently with `read` or `write` can lead to unexpected results.

Signature

```
public
void clear()
```

isEmpty

Queries the queue to determine if `this` is empty.

Note: This method needs to be used with care since the state of the queue may change while the method is in progress or after it has returned.

Signature

```
public
boolean isEmpty()
```

Returns

`true` when `this` is empty; `false` when `this` is not empty.

isFull

Queries the system to determine if `this` is full.

Note: This method needs to be used with care since the state of the queue may change while the method is in progress or after it has returned.

Signature

```
public
boolean isFull()
```

Returns

`true` when `this` is full; `false` when `this` is not full.

read

Reads the least recently inserted element from the queue and returns it as the result, unless the queue is empty. When the queue is empty, `null` is returned.

Signature

```
public
java.lang.Object read()
```

Returns

The `java.lang.Object` read, or else `null` when `this` is empty.

size

Queries the queue to determine the number of elements in `this`.

Note: This method needs to be used with care since the state of the queue may change while the method is in progress or after it has returned.

Signature

```
public
int size()
```

Returns

The number of positions in `this` occupied by elements that have been written but not yet read.

waitForData

When `this` is empty block until a writer inserts an element.

Note: When there is a single reader and no asynchronous invocation of `clear`, then it is safe to invoke `read` after `waitForData` and know that `read` will find the queue non-empty.

Implementation note: To avoid reader and writer synchronizing on the same object, the reader should not be notified directly by a writer. (This is the issue that the non-wait queue classes are intended to solve).

Available since RTSJ 1.0.1 `InterruptedException` **was added to the throws clause.**

Signature

```
public
void waitForData()
throws InterruptedException
```

Throws

UnsupportedOperationException when `this` has not been constructed with `notify` set to `true`.

InterruptedException when the thread is interrupted by `interrupt()` or `AsynchronouslyInterruptedException.fire()`[12] during the time between calling this method and returning from it.

write(Object)

A synchronized and blocking write. This call blocks on queue full and will wait until there is space in the queue.

Available since RTSJ 1.0.1 The return type is changed to void since it *always* returned `true`**, and** `InterruptedException` **was added to the throws clause.**

Signature

```
public synchronized
void write(Object object)
throws InterruptedException
```

[12]Section 8.4.2.1.2

Parameters

object The `java.lang.Object` that is placed in the queue.

Throws

InterruptedException when the thread is interrupted by `interrupt()` or `AsynchronouslyInterruptedException.fire()`[13] during the time between calling this method and returning from it.

MemoryScopeException when a memory access error or illegal assignment error would occur while storing `object` in the queue.

7.3.1.6 WaitFreeWriteQueue

A queue that can be non-blocking for producers. The `WaitFreeWriteQueue` class is intended for single-writer multiple-reader communication, although it may also be used (with care) for multiple writers. A *writer* is generally an instance of `NoHeapRealtimeThread`[14], and the *readers* are generally regular Java threads or heap-using realtime threads or schedulables. Communication is through a bounded buffer of Objects that is managed first-in-first-out. The principal methods for this class are write and read

- The `write` method appends a new element onto the queue. It is not synchronized, and does not block when the queue is full (it returns `false` instead). Multiple writer threads or schedulables are permitted, but when two or more threads intend to write to the same `WaitFreeWriteQueue` they will need to arrange explicit synchronization.
- The `read` method removes the oldest element from the queue. It is synchronized, and will block when the queue is empty. It may be called by more than one reader, in which case the different callers will read different elements from the queue.

`WaitFreeWriteQueue` is one of the classes allowing `NoHeapRealtimeThreads` and regular Java threads to synchronize on an object without the risk of a `NoHeapRealtimeThread` incurring Garbage Collector latency due to priority inversion avoidance management.

Incompatibility with V1.0: Three exceptions previously thrown by the constructor have been deleted from the `throws` clause. These are

- `java.lang.IllegalAccessException`,
- `java.lang.ClassNotFoundException`, and
- `java.lang.InstantiationException`.

[13] Section 8.4.2.1.2

[14] Section 15.3.2.1

Including these exceptions on the `throws` clause was an error. Their deletion may cause compile-time errors in code using the previous constructor. The repair is to remove the exceptions from the `catch` clause around the constructor invocation.

Inheritance

java.lang.Object
javax.realtime.WaitFreeWriteQueue

7.3.1.6.1 Constructors

WaitFreeWriteQueue(Runnable, Runnable, int, MemoryArea)

Constructs a queue in `memory` with an unsynchronized and nonblocking `write()` method and a synchronized and blocking `read()` method.

The `writer` and `reader` parameters, when non-null, are checked to insure that they are compatible with the `MemoryArea` specified by `memory` (when non-null.) When `memory` is `null` and both Runnables are non-null, the constructor will select the nearest common scoped parent memory area, or when there is no such scope it will use immortal memory. When all three parameters are `null`, the queue will be allocated in immortal memory.

`reader` and `writer` are not necessarily the only threads or schedulables that will access the queues; moreover, there is no check that they actually access the queue at all.

Note: that the wait free queue's internal queue is allocated in `memory`, but the memory area of the wait free queue instance itself is determined by the current allocation context.

Signature

```
public
  WaitFreeWriteQueue(Runnable writer, Runnable reader, int maximum, MemoryArea memor
```

Parameters

writer An instance of `Thread`, a schedulable object, or `null`.
reader An instance of `Thread`, a schedulable object, or `null`.
maximum The maximum number of elements in the queue.

memory The `MemoryArea`[15] in which this object and internal elements are allocated.

Throws

IllegalArgumentException when an argument holds an invalid value. The `writer` argument must be `null`, a reference to a `Thread`, or a reference to a schedulable (a `RealtimeThread`, or an `AsyncEventHandler`.) The `reader` argument must be `null`, a reference to a `Thread`, or a reference to a schedulable. The `maximum` argument must be greater than zero.

MemoryScopeException when either `reader` or `writer` is non-null and the `memory` argument is not compatible with `reader` and `writer` with respect to the assignment and access rules for memory areas.

InaccessibleAreaException when `memory` is a scoped memory that is not on the caller's scope stack.

WaitFreeWriteQueue(int, MemoryArea)

Constructs a queue containing up to `maximum` elements in `memory`. The queue has an unsynchronized and nonblocking `write()` method and a synchronized and blocking `read()` method.

Note: that the wait free queue's internal queue is allocated in `memory`, but the memory area of the wait free queue instance itself is determined by the current allocation context.

Available since RTSJ 1.0.1

Signature

```
public
  WaitFreeWriteQueue(int maximum, MemoryArea memory)
```

Parameters

maximum The maximum number of elements in the queue.

memory The `MemoryArea`[16] in which this object and internal elements are allocated.

Throws

IllegalArgumentException when the `maximum` argument is less than or equal to zero, or `memory` is `null`.

[15] Section 11.4.3.6

[16] Section 11.4.3.6

InaccessibleAreaException when `memory` is a scoped memory that is not on the caller's scope stack.

WaitFreeWriteQueue(int)

Constructs a queue containing up to `maximum` elements in immortal memory. The queue has an unsynchronized and nonblocking `write()` method and a synchronized and blocking `read()` method.
Available since RTSJ 1.0.1

Signature

```
public
  WaitFreeWriteQueue(int maximum)
```

Parameters

maximum The maximum number of elements in the queue.

Throws

IllegalArgumentException when the `maximum` argument is less than or equal to zero.

7.3.1.6.2 Methods

---

clear

Sets `this` to empty.

Signature

```
public
void clear()
```

isEmpty

Queries the system to determine if `this` is empty.

Note: This method needs to be used with care since the state of the queue may change while the method is in progress or after it has returned.

Signature

```
public
boolean isEmpty()
```

Returns

True, when `this` is empty. False, when `this` is not empty.

isFull

Queries the system to determine if `this` is full.

Note: This method needs to be used with care since the state of the queue may change while the method is in progress or after it has returned.

Signature

```
public
boolean isFull()
```

Returns

True, when `this` is full. False, when `this` is not full.

read

A synchronized and possibly blocking operation on the queue.

Available since RTSJ 1.0.1 Throws `InterruptedException`

Signature

```
public synchronized
java.lang.Object read()
throws InterruptedException
```

Throws

InterruptedException when the thread is interrupted by `interrupt()` or `AsynchronouslyInterruptedException.fire()`[17] during the time between calling this method and returning from it.

Returns

[17]Section 8.4.2.1.2

The `Object` least recently written to the queue. When `this` is empty, the calling thread or schedulable objects blocks until an element is inserted; when it is resumed, `read` removes and returns the element.

size

Queries the queue to determine the number of elements in `this`.

Note: This method needs to be used with care since the state of the queue may change while the method is in progress or after it has returned.

Signature

```
public
int size()
```

Returns

The number of positions in `this` occupied by elements that have been written but not yet read.

force(Object)

Unconditionally insert `object` into `this`, either in a vacant position or else overwriting the most recently inserted element. The `boolean` result reflects whether, at the time that `force()` returns, the position at which `object` was inserted was vacant (`false`) or occupied (`true`).

Signature

```
public
boolean force(Object object)
```

Parameters

object A non-null `java.lang.Object` to insert.

Throws

MemoryScopeException when a memory access error or illegal assignment error would occur while storing `object` in the queue.

IllegalArgumentException when `object` is `null`.

Returns

`true` when `object` has overwritten an element that was occupied when the function returns; `false` otherwise (it has been inserted into a position that was vacant when the function returns)

write(Object)

Inserts `object` into `this` when `this` is non-full and otherwise has no effect on `this`; the `boolean` result reflects whether `object` has been inserted. When the queue was empty and one or more threads or schedulables were waiting to read, then one will be awakened after the write. The choice of which to awaken depends on the involved scheduler(s).

Signature

```
public
boolean write(Object object)
```

Parameters

object A non-null `java.lang.Object` to insert.

Throws

MemoryScopeException when a memory access error or illegal assignment error would occur while storing `object` in the queue.

IllegalArgumentException when `object` is `null`.

Returns

`true` when the queue was non-full; `false` otherwise.

7.4 Rationale

Java's rules for `synchronized` code provide a means for mutual exclusion but do not prevent unbounded priority inversions and thus are insufficient for realtime applications. This specification strengthens the semantics for `synchronized` code by mandating priority inversion control, in particular by furnishing classes for priority inheritance and priority ceiling emulation. Priority inheritance is more widely implemented in realtime operating systems and thus is required and is the initial default mechanism in this specification.

Since the same object may be accessed from synchronized code by both a `NoHeapRealtimeThread` and an arbitrary thread or schedulable object, unwanted dependencies may result. To avoid this problem, this specification provides three wait-free queue classes as an alternative means for safe, concurrent data accesses without priority inversion.

Chapter 8

Asynchrony

8.1 Overview

One of the most important aspects of this specification is the support for asynchronous control flow. Mechanisms are provided for both starting a task asynchronously and interrupting the execution of a thread or other task. This specifications provides mechanisms that

- bind the execution of program logic to the occurrence of internal and external events;
- enable asynchronous transfer of control; and
- facilitate the asynchronous termination of realtime threads.

This specification provides several facilities for arranging asynchronous control of execution. These facilities fall into two main categories: asynchronous event handling and asynchronous transfer of control, which includes realtime thread termination.

Asynchronous event handling is captured by the classes and subclasses of `AbstractAsyncEvent` (AE), `AbstractAsyncEventHandler` (AEH) and `AbstractBoundAsyncEventHandler`. An AE is an object used to direct event occurrences to asynchronous event handlers. An event occurrence may be initiated by application logic, by mechanisms internal to the `RTSJ` implementation (see the handlers in `PeriodicParameters`), or by some external input such as a clock, a signal, or an interrupt.

An asynchronous event occurrence is initiated in program logic by the invocation of the `fire` method of an AE. The `fire` method dispatches all handlers associated with its event. This means that dispatching occurs in the execution context of the caller.

An asynchronous events that is initiated from an external source has additional requirements and hence additional API features. These features are captured by the ActiveEvent interface. Since external events do not have a full execution context of their own, this category of events must provide an alternate execution context. In order to give the programmer control over this execution context, the specification defines abstract class ActiveEventDispatcher to provide execution context for dispatching. Subclasses provide a `trigger` method for initiating dispatching. Triggering simply informs this execution context to start dispatching. The `trigger` method is not defined in `ActiveEventDispatcher`, since some classes need a `trigger` methods with an argument and others do not. The types of `ActiveEvent` supported are described in subsequent chapters.

An AEH is a schedulable embodying code that is released for execution in response to the occurrence of an associated event. Each AEH behaves as if it is executed by a `RealtimeThread` except that it is not permitted to use the `waitForNextPeriod()`, `waitForNextPeriodInterruptible()`, `waitForNextRelease()`, `waitForNextReleaseInterruptible()` methods, and it is treated as having a null thread group in all cases. There is not necessarily a separate realtime thread for each AEH, but the server realtime thread (returned by `currentRealtimeThread()`) remains constant during each execution of the `run()` method. The class `AbstractBoundAsyncEventHandler` extends `AbstractAsyncEventHandler` and ensures that a handler has a dedicated server realtime thread (a server thread is associated with one and only one bound AEH for the lifetime of that AEH). An event count (called `fireCount`) is maintained so that a handler can cope with event bursts—situations where an event occurs more frequently than its handler can respond.

The `interrupt()` method in `java.lang.Thread` provides rudimentary asynchronous communication by setting a pollable/resettable flag in the target thread, and by throwing a synchronous exception when the target thread is blocked at an invocation of `wait()`, `sleep()`, or `join()`. This specification extends the effect of `Thread.interrupt()` by adding an overridden version in `RealtimeThread`, offering a more comprehensive and non-polling asynchronous execution control facility. It is based on throwing and propagating exceptions that, though asynchronous, are deferred where necessary in order to avoid data structure corruption. The main elements of ATC are embodied in the class `AsynchronouslyInterruptedException`, its subclass `Timed`, the interface `Interruptible`, and in the semantics of the `interrupt` method in `RealtimeThread`.

A method indicates its eligibility to be asynchronously interrupted by including the checked exception `AsynchronouslyInterruptedException` in its `throws` clause. If a schedulable is asynchronously interrupted while executing such a method, then an AIE will be delivered as soon as the schedulable is outside of a section in which ATC is deferred. Several idioms are available for handling an AIE, giving the programmer the choice of using `catch` clauses and a low-level mechanism with specific control over propagation, or a higher-level facility that allows specifying the interruptible code, the handler, and the result retrieval as separate methods.

8.2 Definitions

The following terms and abbreviations will be used.

AE—Asynchronous Event. An instance of one of the subclasses of the `javax.realtime.Abstract` class.

AEH—Asynchronous Event Handler. An instance of one of the subclasses of the AbstractAsyncEventHandler class.

Bound AEH — Bound Asynchronous Event Handler. An instance of one of the subclasses of the AbstractBoundAsyncEventHandler class.

ATC — Asynchronous Transfer of Control.

AIE — Asynchronously Interrupted Exception. An instance of the `javax.realtime.Asynchronou` class (a subclass of `java.lang.InterruptedException`).

AI-method - Asynchronously Interruptible method. A method or constructor that includes `AsynchronouslyInterruptedException` explicitly (that is not a subclass of `AsynchronouslyInterruptedException`) in its throws clause.

Lexical Scope [of a method, constructor, or statement]. The textual region within the constructor, method, or statement, excluding the code within any class declarations, and the code within any class instance creation expressions for anonymous classes, contained therein. The lexical scope of a construct does not include the bodies of any methods or constructors that this code invokes.

ATC-deferred section. A synchronized statement, a static initializer or any method or constructor without `AsynchronouslyInterruptedException` in its throws clause. As specified in the introduction to Chapter 8 in *Java Language Specification*, a synchronized method is equivalent to a non-synchronized method with the body of the method contained in a synchronized statement. Thus, a synchronized AI method behaves like an AI method containing only an ATC-deferred statement.

Interruptible blocking methods. The `RTSJ` and standard Java methods that are explicitly interruptible by an AIE. The interruptible blocking methods comprise

- `HighResolutionTime.waitForObject()`,
- `Object.wait()`,
- `Thread.sleep()`,
- `RealtimeThread.sleep()`,
- `Thread.join()`,
- `ScopedMemory.join()`,
- `ScopedMemory.joinAndEnter()`,
- `RealtimeThread.waitForNextPeriodInterruptible()`,
- `RealtimeThread.waitForNextReleaseInterruptible()`,
- `WaitFreeWriteQueue.read()`,
- `WaitFreeReadQueue.waitForData()`,
- `WaitFreeReadQueue.write()`,
- `WaitFreeDequeue.blockingRead()`,

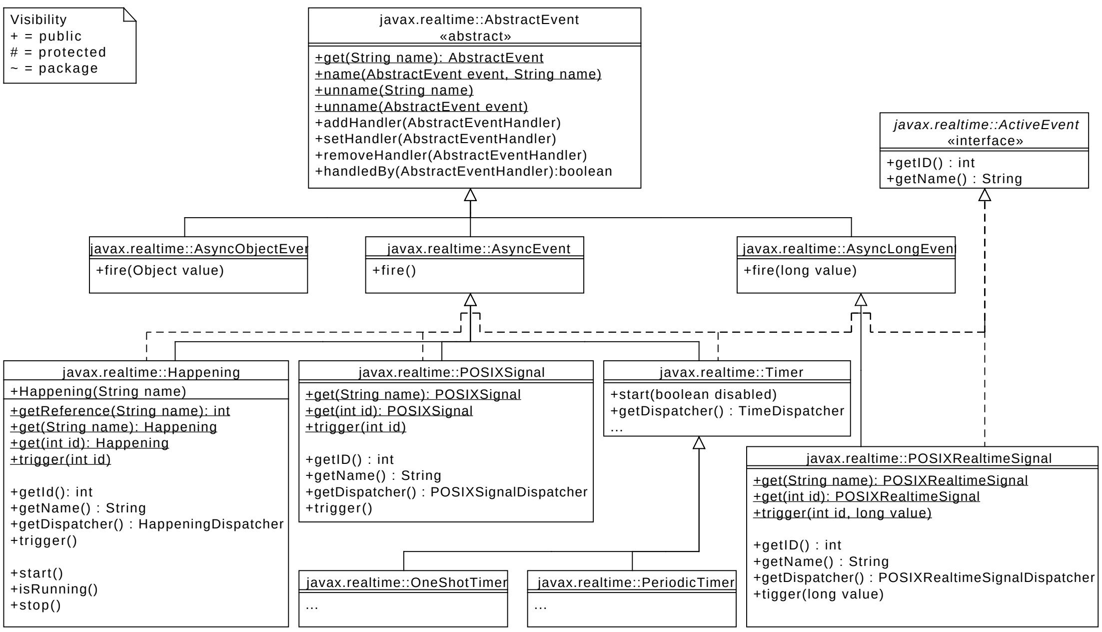
- `WaitFreeDequeue.blockingWrite()`

and their overloaded forms.

8.3 Semantics

Asynchronous events and event handlers are required in the base module, whereas asynchronous transfere of control is optional. Basic event types are passive: they are not directly associated with a thread of control. They are intended to be fired programmatically. Handelling external events, such as clocks (see Chapter 10) and happening (see Chapter 12), requires execution support. The `ActiveEvent` interface is provided to mark these and provide additional execution semantics. Figure 8.1 illustrates the event hierarchy.

Figure 8.1: The Event Class Hierarchy



8.3.1 Asynchronous Events and their Handlers

This following list establishes the semantics that are applicable to asynchronous events and their handlers. Semantics that apply to particular classes, constructors, methods, and fields will be found in the class description and the constructor, method, and field detail sections.

1. When an asynchronous event occurs (by either program logic or by the triggering of a happening), its attached handlers (that is, AEHs that have been added to the AE by the execution of `addHandler()`) are released for execution. Every occurrence of an event increments the `fireCount` in each attached handler. Handlers may elect to execute logic for each occurrence of the event or not.
2. The release of attached handlers occurs in execution eligibility order (priority order, from highest to lowest, with the default `PriorityScheduler`) and at the active priority of the schedulable that invoked the `fire` method. The release of handlers resulting from a happening or a timer must begin within a bounded time (ignoring time consumed by unrelated activities in the system). This worst-case response interval must be documented for some reference architecture.
3. The release of attached handlers is an atomic operation with respect to adding and removing handlers.
4. The logical release of an attached handler may occur before the previous release has completed.
5. A deadline may be associated with each logical release of an attached handler. The deadline is relative to the occurrence of the associated event.
6. AEs and AEHs may be created and used by any program logic within the constraints of the memory assignment rules.
7. More than one AEH may be added to an AE. However, adding an AEH to an AE has no effect if the AEH is already attached to the AE.
8. The same AEH may be added to more than one AE.
9. By default all AEHs are considered to be daemons (the daemon status being set by their constructors). An AEH can be set to have a non daemon status after it has been created and before it has been attached to an AE.
10. The object returned by `currentRealtimeThread()` while an AEH is running shall behave with respect to memory access and assignment rules as if it were allocated in the same memory area as the AEH.
11. System-related termination activity (such as execution of finalizers for scoped objects in scopes that become unreferenced) triggered when an AEH becomes unfireable is not subject to cost enforcement or deadline miss detection.
12. AEs and AEHs behave effectively as if changes to an AEH's fireability are contained in synchronized blocks, and the AEH holds that lock while it is in the process of becoming non-fireable.

An `RTSJ` program terminates when and only when

- all non-daemon threads (either regular Java threads or realtime threads) are terminated,
- the `fireCounts` of all nondaemon bound AEHs or nondaemon AEHs are zero

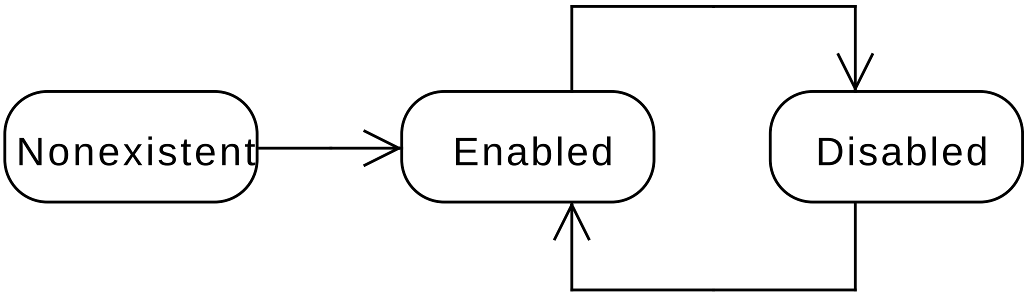
and all releases are completed, and

- there are no nondaemon Bound AEHs or AEHs attached to timers or async events associated with happenings.

Though dispatchers have a thread, this thread is a daemon thread and does itself not hinder termination.

`AbstractAsyncEvent` provides two basic states: enabled and disabled. In the enabled state, fire causes all associated handlers to be dispatched, whereas fire does nothing when the event is disabled. Figure 8.2 illustrates this state space.

Figure 8.2: States of a Simple AbstractAsyncEvent



8.3.2 Active Events and Dispatching

Active events refine the semantics of AbstractAsyncEventHandler with the addition of execution semantics to support second level interrupt handling. The fire method of an event runs in the Java execution context of the caller. For events that represent external signals, whether a certain time is reached or something has occurred, there may not be a Java execution context, or at least that context is of necessity limited and often of needs to have a very short duration; dispatching an unlimited number of handlers is not acceptable. They require an additional execution context for releasing handlers.

In order to be able to distinguish between events that are caused to be fired by an outside mechanism from those that are fired from another thread, the former extend the `ActiveEvent` interface. Since the `trigger` methods may vary in the number of their arguments depending on the type of event, each class implementing

`ActiveEvent` must provide its own `trigger` method for initiating the handler release by releasing another execution context. Each method must act as if it calls the `fire` method on its event and then terminates. Hence `trigger` has the same functional behavior as `fire` but runs in this other execution context.

This extra execution context is exposed to the user as an `ActiveEventDispatcher`. There is a active event dispatcher for each kind of active event. The programmer does not need to write a dispatcher, but just creates the one of the corresponding type. The programmer does determine the priority and the affinity of a dispatcher, as well as determine the mapping between dispatchers and events.

Each event has a single dispatcher, but a dispatcher may serve many events. As with `fire`, the dispatcher releases handlers in reverse priority order, i.e., from highest to lowest. This enables the programmer to control the number of these execution contexts and still optimize how handlers are released.

The state space of an `ActiveEvent` is an extention of the state space for an `AbstractAsyncEvent` depicted in Figure 8.2. `ActiveEvent` adds the notion of active and inactive on top of enabled and disabled, as depicted in Figure 8.3. Note the enabled-disabled distinction is only splits the active state. The inactive state is by definition disabled.

8.3.3 Asynchronous Transfer of Control

Asynchronously interrupting a schedulable consists of the following activities.

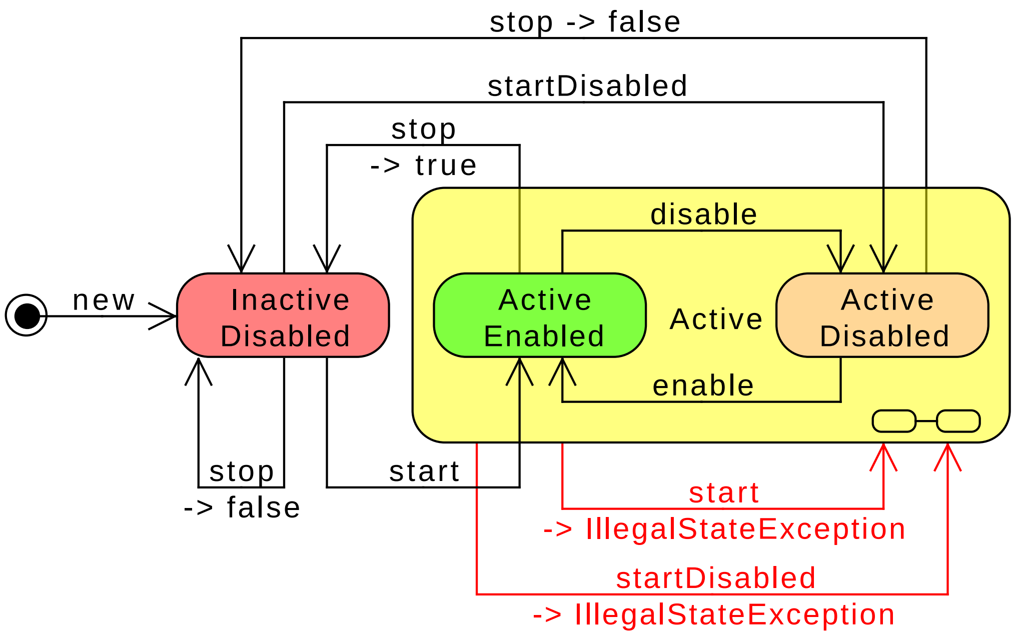
- **Generation** of an asynchronous interrupt exception — this is the event in the underlying system that makes the AIE available to the program.
- **Delivery** of the asynchronous interrupt exception to the target schedulable—this is the action that invokes the search for and execution of an appropriate handler.

Between the generation and delivery, the asynchronous interrupt exception is held *pending*. After delivery, the AIE remains pending until it is **cleared** by the program logic using `clear()` or `doInterruptible()`.

This following list establishes the semantics that are applicable to ATC. Semantics that apply to particular classes, constructors, methods, and fields will be found in the class description and the constructor, method, and field detail sections.

1. An AIE is generated for a given schedulable, when the `fire()` method is called on an AIE for which the schedulable object is executing within the `doInterruptible()` method, or the `RealtimeThread.interrupt()` method is called; the latter is effectively called when an AIE is generated by internal virtual machine mechanisms (such as an interrupt I/O protocol) that are asynchronous to the execution of program logic which is the target of the AIE. A generated AIE becomes pending upon generation and remains pending until explicitly cleared or replaced by another AIE.

Figure 8.3: States of an ActiveEvent



2. The `RealtimeThread.interrupt()` method throws the generic AIE at the target realtime thread and has the behaviors defined for `Thread.interrupt()`. This is the only interaction between the ATC mechanism and the conventional `interrupt()` mechanism.
3. An AIE is delivered to a schedulable when it is executing in an AI-method except as indicated below.
4. The generation of an AIE through the `fire()` mechanism behaves as if it set an asynchronously-interrupted status in the schedulable. If the schedulable is blocked within an interruptible blocking method, or invokes an interruptible blocking method, when this asynchronously-interrupted status is set, then the invocation immediately completes by throwing the pending AIE and clearing the asynchronously-interrupted status. When a pending AIE is explicitly cleared then the asynchronously-interrupted status is also cleared.
5. Methods which block through mechanisms other than the interruptible blocking methods, (for example, blocking methods in `java.io.*`) must be prevented from blocking indefinitely when invoked from a method with `Asynchronously-`

`InterruptedException` in its `throws` clause. When an AIE is generated and the target schedulable's control is blocked inside one of these methods invoked from an AI-method, the implementation may either unblock the blocked call, raise an `InterruptedIOException` on behalf of the call, or allow the call to complete normally if the implementation determines that the call would eventually unblock.

6. If an AI-method is attempting to acquire an object lock when an associated AIE is generated, the attempt to acquire the lock is abandoned.
7. If control is in the lexical scope of an ATC-deferred section when an AIE (targeted at the executing schedulable) is generated, the AIE is not delivered until the first subsequent attempt to transfer control to code that is not ATC deferred. At that point, control is transferred to the `catch` or `finally` clause of the nearest dynamically-enclosing a `try` statement that has a handler for the generated AIE's (that is a handler naming the AIE's class or any of its superclasses, or a `finally` clause) and which is in an ATC-deferred section. Intervening handlers and `finally` clauses that are not in ATC-deferred sections are not executed, but object locks are released.
   See section 11.3 of *The Java Language Specification* second edition for an explanation of the terms, *dynamically enclosing* and *handler*. The `RTSJ` uses those JLS definitions unaltered. Note, if synchronized code is abandoned as a result of this control transfer, the associated locks are released.
8. Constructors are allowed to include `AsynchronouslyInterruptedException` in their `throws` clause and if they do will be asynchronously interruptible under the same conditions as AI methods.
9. Native methods that include `AsynchronouslyInterruptedException` in their `throws` clause have implementation-specific behavior.
10. An implementation must deliver the transfer of control in a schedulable that is subject to asynchronous interruption (in an AI-method but not in a synchronized block) within a bounded execution time of that schedulable. This worst-case response interval must be documented for some reference architecture.
11. Instances of the `Timed` class logically have an associated timer. When the timer fires, the schedulable executing the instance's `doInterruptible` method must have the AIE generated within a bounded execution time of the schedulable. This worst-case response interval must be documented for some reference architecture.
12. An AIE only has the semantics defined here if it originates with the `AsynchronouslyInterruptedException.fire()` method, the `RealtimeThread.interrupt()` method or from within the realtime VM. If an AIE is thrown from program logic using the Java throw statement, it acts the same as throwing any other

instance of a subclass of `Exception`, it is processed as a normal exception, and has no affect on the pending state of any AIE, and no affect on the firing of the AIE concerned.

8.3.3.1 Summary of ATC Operation

The `RTSJ`'s approach to ATC is designed to follow the above principles. It is based on exceptions and is an extension of the current Java language rules for `java.lang.Thread.interrupt()`. In summary, ATC works as follows:

If `so` is an instance of a schedulable and the `interrupt()` method is called on the realtime thread associated with that object (in this context, the associated realtime thread of an AEH is the realtime thread returned by a call of the `RealtimeThread.currentRealtimeThread()` method by that AEH) then:

- If control is in an ATC-deferred section, then the AIE remains in a pending state.
- If control is not in an ATC-deferred section, then control is transferred to the `catch` or `finally` clause of the nearest dynamically-enclosing a `try` statement that has a handler for the generated AIE's (that is a handler naming the AIE's class or any of its superclasses, or a `finally` clause) and which is in an ATC-deferred section. Intervening handlers and `finally` clauses that are not in ATC-deferred sections are not executed, but objects locks are released. See section 11.3 of *The Java Language Specification* second edition for an explanation of the terms, *dynamically enclosing* and *handles*. The `RTSJ` uses those definitions unaltered.
- If control is in an interruptible blocking method the schedulable object is awakened and the generated AIE (which is a subclass of `InterruptedException)` is thrown with regular Java semantics (the AIE is still marked as pending). Then ATC follows option 1, or 2 as appropriate.
- If control is in an ATC-deferred section, control continues normally until the first attempt to return to an AI method or invoke an AI method or exit a synchronized block within an AI method. Then ATC follows option 1, or 2 as appropriate.
- If control is transferred from an ATC-deferred section to an AI method through the action of propagating an exception and if an AIE is pending then when the transition to the AI-method occurs, the thrown exception is discarded and replaced by the AIE.

An AIE may be generated while another AIE is pending. Because AI code blocks are nested by method invocation (a stack-based nesting) there is a natural precedence among active instances of AIE. Let AIE_0 be the AIE raised when the `RealtimeThread.interrupt()` method is invoked and AIE_i ($i = 1, ..., n$, for n unique instances of AIE) be the AIE generated when AIE. `fire()` is invoked. In the following, the phrase "a frame deeper on the stack than this frame" refers to a method nearer to the current stack frame. The phrase "a frame shallower on the stack than this frame" refers to a method further from the current stack frame.

- If the current AIE is an AIE_0 and the new AIE is an AIE_x associated with any frame on the stack then the new AIE (AIE_x) is discarded.
- If the current AIE is an AIE_x and the new AIE is an AIE_0, then the current AIE (AIE_x) is replaced by the new AIE (AIE_0).
- If the current AIE is an AIE_x and the new AIE is an AIE_y from a frame deeper on the stack, then the new AIE (AIE_y) discarded.
- If the current AIE is an AIE_x and the new AIE is an AIE_y from a frame shallower on the stack, the current AIE (AIE_x) is replaced by the new AIE (AIE_y).
- If the current AIE is an AIE_0 and the new AIE is an AIE_0, or if the current AIE is an AIE_x and the new AIE is an AIE_x, the new AIE is discarded.

When clear() or `happened()` is called on a pending AIE or that AIE is superseded by another, the first AIE's pending state is cleared. If the `happened()` method is called on a non-pending AIE the result depends on the value of the `propagate` parameter, as indicated in the "No Match" column of the table below. Clearing a non-pending AIE (with the `clear()` method) has no effect.

propagate	Match	No Match
true	clear the pending AIE, return true	the AIE remains pending, propagate
false	clear the pending AIE, return true	the AIE remains pending, return false

8.4 Package javax.realtime

8.4.1 Interfaces

8.4.1.1 ActiveEvent

This is the interface for defining the active event system. Classes implementing `ActiveEvent` are used to connect events that take place outside the Java virtual machine to RTSJ activities.

When an event takes place outside the Java virtual machine, some event-specific code within the Java virtual machine executes. That code notifies the `ActiveEvent` infrastructure of this event by calling a `trigger` method in the event.

8.4.1.1.1 Methods

isActive

Determine the activation state of this event, i.e., it has been started but not yet stopped again.

Signature

```
public
boolean isActive()
```

Returns

`true` when active, `false` otherwise.

isRunning

Determine the running state of this event, i.e., it is both active and enabled.

Signature

```
public
boolean isRunning()
```

Returns

`true` when active and enabled, `false` otherwise.

start

Start this active event.

Signature

```
public
void start()
throws IllegalStateException
```

Throws

IllegalStateException when this event has already been started.

start(boolean)

Start this active event.

Signature

```
public
void start(boolean disabled)
throws IllegalStateException
```

Parameters

disabled true for starting in a disabled state.

Throws

IllegalStateException when this event has already been started.

stop

Stop this active event.

Signature

```
public
boolean stop()
throws IllegalStateException
```

Throws

IllegalStateException when this event is not running.

Returns

the previous enabled state.

enable

Change the state of the event so that associated handlers are release on fire. Each subclass provides a means of dispatching its handlers when requested. This method enables that request mechanism.

Signature

```
public
void enable()
```

disable

Change the state of the event so that associated handlers are skipped on fire. Each subclass provides a fire method as means of dispatching its handlers when requested. This method disables that request mechanism.

Signature

```
public
void disable()
```

8.4.1.2 BoundAbstractAsyncEventHandler

An marker interface for all schedulables that are bound to a single thread of control. It is required to enable references to all bound handlers.

Interfaces

BoundSchedulable

8.4.1.3 Interruptible

`Interruptible` is an interface implemented by classes that will be used as arguments on the method `doInterruptible()` of `AsynchronouslyInterruptedExcep-`

tion[1] and its subclasses. `doInterruptible()` invokes the implementation of the method in this interface.

8.4.1.3.1 Methods

---

run(AsynchronouslyInterruptedException)

The main piece of code that is executed when an implementation is given to `doInterruptible()`. When a class is created that implements this interface (for example through an anonymous inner class) it must include the `throws` clause to make the method interruptible. When the throws clause is omitted the `run()` method will not be interruptible.

Signature

```
public
void run(AsynchronouslyInterruptedException exception)
throws AsynchronouslyInterruptedException
```

Parameters

exception The AIE object whose doInterruptible method is calling the `run` method. Used to invoke methods on `AsynchronouslyInterruptedException`[2] from within the `run()` method.

interruptAction(AsynchronouslyInterruptedException)

This method is called by the system when the `run()` method is interrupted. Using this, the program logic can determine when the `run()` method completed normally or had its control asynchronously transferred to its caller.

Signature

```
public
void interruptAction(AsynchronouslyInterruptedException
exception)
```

Parameters

[1] Section 8.4.2.1
[2] Section 8.4.2.1

exception The currently pending AIE. Used to invoke methods on `AsynchronouslyInterruptedException`[3] from within the `interruptAction()` method.

8.4.2 Exceptions

8.4.2.1 AsynchronouslyInterruptedException

A special exception that is thrown in response to an attempt to asynchronously transfer the locus of control of a schedulable.

A schedulable that is executing a method or constructor, which is declared with an `AsynchronouslyInterruptedException`[4] in its `throws` clause, can be asynchronously interrupted except when it is executing in the lexical scope of a synchronized statement within that method/constructor. As soon as the schedulable object leaves the lexical scope of the method by calling another method/constructor it may be asynchronously interrupted when the called method/constructor is asynchronously interruptible. (See this chapter's introduction section for the detailed semantics).

The asynchronous interrupt is generated for a realtime thread, `t`, when the `t.interrupt()` method is called or the `fire`[5] method is called of an AIE for which t has a `doInterruptible` method call in progress.

The interrupt is generated for an AEH (or BAEH), `h`, when the `fire`[6] method is called of an AIE for which `h` has a `doInterruptible` method call in progress.

When an asynchronous interrupt is generated when the target realtime thread/schedulable is executing within an ATC-deferred section, the asynchronous interrupt becomes pending. A pending asynchronous interrupt is delivered when the target realtime thread/schedulable next attempts to enter asynchronously interruptible code.

Asynchronous transfers of control (ATCs) are intended to allow long-running computations to be terminated without the overhead or latency of polling with **name**.

When `RealtimeThread.interrupt`[7], or `AsynchronouslyInterruptedException.fire()` is called, the `AsynchronouslyInterruptedException` is compared against any currently pending `AsynchronouslyInterruptedException` on the schedulable. When there is none, or when the depth of the `AsynchronouslyInterruptedException` is less than the currently pending `AsynchronouslyInterruptedException`; (i.e., it is targeted at a less deeply nested method call), the new `AsynchronouslyInterr-`

[3] Section 8.4.2.1
[4] Section 8.4.2.1
[5] Section 8.4.2.1.2
[6] Section 8.4.2.1.2
[7] Section 5.3.2.2.2

`uptedException` becomes the currently pending `AsynchronouslyInterruptedException` and the previously pending `AsynchronouslyInterruptedException` is discarded. Otherwise, the new `AsynchronouslyInterruptedException` is discarded.

When an `AsynchronouslyInterruptedException` is caught, the catch clause may invoke the `clear()` method on the `AsynchronouslyInterruptedException` in which it is interested to see if the exception matches the pending `AsynchronouslyInterruptedException`. When so, the pending `AsynchronouslyInterruptedException` is cleared for the schedulable and `clear` returns true. Otherwise, the current AIE remains pending and `clear` returns false.

`RealtimeThread.interrupt()` generates a system-wide generic `AsynchronouslyInterruptedException` which will always propagate outward through interruptible methods until the generic `AsynchronouslyInterruptedException` is identified and handled. The pending state of the generic AIE is per-schedulable object.

Other sources (e.g., `AsynchronouslyInterruptedException.fire()` and `Timed`[8]) will generate specific instances of `AsynchronouslyInterruptedException` which applications can identify and thus limit propagation.

`AsyncEventHandler`[9] objects should interact with the ATC mechanisms via the `Interruptible`[10] interface.

Inheritance

java.lang.Object
  java.lang.Throwable
    java.lang.Exception
      java.lang.InterruptedException
        javax.realtime.AsynchronouslyInterruptedException

8.4.2.1.1 Constructors

AsynchronouslyInterruptedException

Create an instance of `AsynchronouslyInterruptedException`.

Signature

```
public
  AsynchronouslyInterruptedException()
```

[8] Section 8.4.2.2
[9] Section 8.4.3.5
[10] Section 8.4.1.3

8.4.2.1.2 Methods

getGeneric

Gets the singleton system generic `AsynchronouslyInterruptedException` that is generated when `RealtimeThread.interrupt()`[11] is invoked.

Signature

```
public static
javax.realtime.AsynchronouslyInterruptedException getGeneric()
```

Throws

IllegalThreadStateException when the current thread is a Java thread.

Returns

The generic `AsynchronouslyInterruptedException`.

enable

Enable the throwing of this exception. This method is valid only when the caller has a call to `doInterruptible()` in progress. When invoked when no call to `doInterruptible()` is in progress, `enable` returns false and does nothing.

Signature

```
public
boolean enable()
```

Returns

True when `this` was disabled before the method was called and the call was invoked whilst the associated `doInterruptible()` is in progress. False: otherwise.

disable

[11] Section 5.3.2.2.2

Disable the throwing of this exception. When the `fire`[12] method is called on `this` AIE whilst it is disabled, the fire is held pending and delivered as soon as the AIE is enabled and the interruptible code is within an AI-method. When an AIE is pending when the associated disable method is called, the AIE remains pending, and is delivered as soon as the AIE is enabled and the interruptible code is within an AI-method.

This method is valid only when the caller has a call to `doInterruptible()` in progress. If invoked when no call to `doInterruptible()` is in progress, `disable` returns false and does nothing.

Note: disabling the genericAIE associated with a realtime thread only affects the firing of that AIE. When the genericAIE is generated by the `RealtimeThread.interrupt()`[13] mechanism, the AIE is delivered (unless the `Interruptible` code is in an AI-deferred region, in which case it is marked as pending and handled in the usual way).

Signature

```
public synchronized
boolean disable()
```

Returns

True when `this` was enabled before the method was called and the call was invoked with the associated `doInterruptible()` in progress. False: otherwise.

isEnabled

Query the enabled status of this exception.

This method is valid only when the caller has a call to `doInterruptible()` in progress. If invoked when no call to `doInterruptible()` is in progress, `enable` returns false and does nothing.

Signature

```
public
boolean isEnabled()
```

Returns

True when this is enabled and the method call was invoked in the context of the associated `doInterruptible()`. False otherwise.

[12] Section 8.4.2.1.2
[13] Section 5.3.2.2.2

fire

Generate this exception when its `doInterruptible()` has been invoked and not completed. When `this` is the only outstanding AIE on the schedulable object that invoked this AIE's `doInterruptible(Interruptible)`[14] method, this AIE becomes that schedulable's current AIE. Otherwise, it only becomes the current AIE when it is at a less deep level of nesting compared with the current outstanding AIE.

Signature

```
public
boolean fire()
```

Returns

True when `this` is not disabled and it has an invocation of a `doInterruptible()` in progress and there is no outstanding fire request. False otherwise.

doInterruptible(Interruptible)

Executes the `run()` method of the given `Interruptible`[15]. This method may be on the stack in exactly one `Schedulable`[16] object. An attempt to invoke this method in a schedulable while it is on the stack of another or the same schedulable will cause an immediate return with a value of false.

The `run` method of given `Interruptible` is always entered with the exception in the enabled state, but that state can be modified with `enable()`[17] and `disable()`[18] and the state can be observed with `isEnabled()`[19].

This AIE is cleared on return from `doInterruptible()`.

Signature

```
public
boolean doInterruptible(Interruptible logic)
```

Parameters

logic An instance of an `Interruptible`[20] whose `run()` method will be called.

Throws

[14] Section 8.4.2.1.2
[15] Section 8.4.1.3
[16] Section 6.4.1.2
[17] Section 8.4.2.1.2
[18] Section 8.4.2.1.2
[19] Section 8.4.2.1.2
[20] Section 8.4.1.3

IllegalThreadStateException when called from a Java thread.
IllegalArgumentException when `logic` is `null`.

Returns

True when the method call completed normally. False when another call to `doInterruptible` has not completed.

clear

Atomically see if `this` is pending on the currently executing schedulable, and when so, make it non-pending.
Available since RTSJ 1.0.1

Signature

```
public
boolean clear()
```

Throws

IllegalThreadStateException when called from a Java thread.

Returns

True when `this` was pending. False when `this` was not pending.

8.4.2.2 Timed

Open issue: The fact that Timed extends AIE confuses the javadoc. It prints an Exceptions heading

Create a scope in a `Schedulable`[21] object which will be asynchronously interrupted at the expiration of a timer. This timer will begin measuring time at some point between the time `doInterruptible()` is invoked and the time the `run()` method of the `Interruptible` object is invoked. Each call of `doInterruptible()` on an instance of `Timed` will restart the timer for the amount of time given in the constructor or the most recent invocation of `resetTime()`. The timer is cancelled when it has not expired before the `doInterruptible()` method has finished.

All memory use of an instance of `Timed` occurs during construction or the first invocation of `doInterruptible()`. Subsequent invocations of `doInterruptible()` do not allocate memory.

[21] Section 6.4.1.2

When the timer fires, the resulting AIE will be generated for the schedulable within a bounded execution time of the targeted schedulable.

Typical usage: `new Timed(T).doInterruptible(interruptible);` **End of open issue**

Inheritance

java.lang.Object
  java.lang.Throwable
    java.lang.Exception
      java.lang.InterruptedException
        javax.realtime.AsynchronouslyInterruptedException
          javax.realtime.Timed

8.4.2.2.1 Constructors

Timed(javax.realtime.HighResolutionTime<T>)

Create an instance of `Timed` with a timer set to `time`. When the `time` is in the past the `AsynchronouslyInterruptedException`[22] mechanism is activated immediately after or when the `doInterruptible()` method is called.

Signature

```
public
  Timed(javax.realtime.HighResolutionTime<T> time)


 throws IllegalArgumentException, UnsupportedOperationException
```

Parameters

time When `time` is a `RelativeTime`[23] value, it is the interval of time between the invocation of `doInterruptible()` and when the schedulable is asynchronously interrupted. When `time` is an `AbsoluteTime`[24] value, the timer asynchronously interrupts at this time (assuming the timer has not been cancelled).

Throws

IllegalArgumentException when `time` is `null`.

[22]Section 8.4.2.1
[23]Section 9.4.1.3
[24]Section 9.4.1.1

UnsupportedOperationException when `time` is not based on a `Clock`[25].

8.4.2.2.2 Methods

---

doInterruptible(Interruptible)

Execute a time-out method. Starts the timer and executes the `run()` method of the given `Interruptible`[26] object.

Signature

```
public
boolean doInterruptible(Interruptible logic)
```

Parameters

logic @inheritDoc

Throws

IllegalArgumentException @inheritDoc
IllegalThreadStateException @inheritDoc

Returns

@inheritDoc

resetTime(javax.realtime.HighResolutionTime<T>)

To set the time-out for the next invocation of `doInterruptible()`.

Signature

```
public
void resetTime(javax.realtime.HighResolutionTime<T> time)
```

Parameters

time This can be an absolute time or a relative time. When `null` or not based on a `Clock`[27], the time-out is not changed.

---

[25]Section 10.4.2.1
[26]Section 8.4.1.3
[27]Section 10.4.2.1

restart(javax.realtime.HighResolutionTime<T>)

Reset the timeout. When this `Timed`[28] instance is executing, adjust the timeout to `time` and restart the timer. When the instance is not executing, adjust the timeout for the next invocation.
Available since RTSJ 2.0

Signature

```
public
void restart(javax.realtime.HighResolutionTime<T> time)
```

Parameters

time The new timeout.

Throws

IllegalArgumentException when `time` is `null` or a relative time less than zero.
UnsupportedOperationException when `time` is not based on a `Clock`[29]

8.4.3 Classes

8.4.3.1 AbstractAsyncEvent

This is the base class for all asynchronous events, where asynchronous is in regards to running code, not external time. This class unifies the original `AsyncEvent`[30] with `AsyncLongEvent`[31] and `AsyncObjectEvent`[32].
Available since RTSJ 2.0

Inheritance

java.lang.Object
  javax.realtime.AbstractAsyncEvent

8.4.3.1.1 Methods

---

[28]Section 8.4.2.2
[29]Section 10.4.2.1
[30]Section 8.4.3.4
[31]Section 8.4.3.6
[32]Section 8.4.3.8

isRunning

Determine the firing state (releasing or skipping) of this event, i.e., whether it is enabled or disabled.

Signature

```
public
boolean isRunning()
```

Returns

`true` when releasing, `false` when skipping.

handledBy(AbstractAsyncEventHandler)

Test to see if the handler given as the parameter is associated with `this`.

Signature

```
public
boolean handledBy(AbstractAsyncEventHandler handler)
```

Parameters

handler The handler to be tested to determine if it is associated with `this`.

Returns

True when the parameter is associated with `this`. False when `handler` is `null` or the parameters is not associated with `this`.

enable

Change the state of the event so that associated handlers are release on fire. Each subclass provides a means of dispatching its handlers when requested. This method enables that request mechanism.

Signature

```
public
void enable()
```

disable

Change the state of the event so that associated handlers are skipped on fire. Each subclass provides a fire method as means of dispatching its handlers when requested. This method disables that request mechanism.

Signature

```
public
void disable()
```

addHandler(AbstractAsyncEventHandler)

Add a handler to the set of handlers associated with this event. An instance of `AsyncEvent` may have more than one associated handler. However, adding a handler to an event has no effect when the handler is already attached to the event.

The execution of this method is atomic with respect to the execution of the `fire()` method.

Since this affects the constraints expressed in the release parameters of an existing schedulable, this may change the feasibility of the current system. This method does not change feasibility set of any scheduler, and no feasibility test is performed.

Note, there is an implicit reference to the handler stored in `this`. The assignment must be valid under any applicable memory assignment rules.

Signature

```
public
void addHandler(AbstractAsyncEventHandler handler)
```

Parameters

handler The new handler to add to the list of handlers already associated with this. When `handler` is already associated with the event, the call has no effect.

Throws

IllegalArgumentException when `handler` is `null` or the handler has `PeriodicParameters`[33]. Only the subclass `PeriodicTimer`[34] is allowed to have handlers with `PeriodicParameters`[35].

IllegalAssignmentError when this `AsyncEvent` cannot hold a reference to `handler`.

setHandler(AbstractAsyncEventHandler)

[33]Section 6.4.2.4
[34]Section 10.4.2.3
[35]Section 6.4.2.4

Associate a new handler with this event and remove all existing handlers. The execution of this method is atomic with respect to the execution of the `fire()` method.

Since this affects the constraints expressed in the release parameters of the existing schedulables, this may change the feasibility of the current system. This method does not change the feasibility set of any scheduler, and no feasibility test is performed.

Signature

```
public
void setHandler(AbstractAsyncEventHandler handler)
```

Parameters

handler The instance of `AbstractAsyncEventHandler`[36] to be associated with `this`. When `handler` is `null` then no handler will be associated with `this`, i.e., behave effectively as if `setHandler(null)` invokes `removeHandler(AbstractAsyncEventHandl`
for each associated handler.

Throws

IllegalArgumentException when `handler` has `PeriodicParameters`[38]. Only the subclass `PeriodicTimer`[39] is allowed to have handlers with `PeriodicParameters`[40].

IllegalAssignmentError when this `AsyncEvent` cannot hold a reference to `handler`.

removeHandler(AbstractAsyncEventHandler)

Remove a handler from the set associated with this event. The execution of this method is atomic with respect to the execution of the `fire()` method.

A removed handler continues to execute until its fireCount becomes zero and it completes.

When `handler` has a scoped non-default initial memory area and execution of this method causes `handler` to become non-fireable, this method shall not return until all related finalization has completed.

[36] Section 8.4.3.2
[37] Section 8.4.3.1.1
[38] Section 6.4.2.4
[39] Section 10.4.2.3
[40] Section 6.4.2.4

Signature

```
public
void removeHandler(AbstractAsyncEventHandler handler)
```

Parameters

handler The handler to be disassociated from `this`. When `null` nothing happens. When the `handler` is not already associated with `this` then nothing happens.

createReleaseParameters

Create a `ReleaseParameters`[41] object appropriate to the release characteristics of this event. The default is the most pessimistic: `AperiodicParameters`[42]. This is typically called by code that is setting up a handler for this event that will fill in the parts of the release parameters for which it has values, e.g., cost. The returned `ReleaseParameters`[43] object is not bound to the event. Any changes in the event's release parameters are not reflected in previously returned objects.

When an event returns `PeriodicParameters`[44], there is no requirement for an implementation to check that the handler is released periodically.

Signature

```
public
javax.realtime.ReleaseParameters createReleaseParameters()
```

Returns

A new `ReleaseParameters`[45] object.

8.4.3.2 AbstractAsyncEventHandler

This is the base class for all asynchronous event handlers, where asynchronous is in regards to running code, not external time. This class unifies the original `Async-EventHandler`[46] with `AsyncLongEventHandler`[47] and `AsyncObjectEventHandler`[48].

[41] Section 6.4.2.8
[42] Section 6.4.2.2
[43] Section 6.4.2.8
[44] Section 6.4.2.4
[45] Section 6.4.2.8
[46] Section 8.4.3.5
[47] Section 8.4.3.7
[48] Section 8.4.3.9

Available since RTSJ 2.0

Inheritance

java.lang.Object
javax.realtime.AbstractAsyncEventHandler

Interfaces

Schedulable

8.4.3.2.1 Constructors

8.4.3.2.2 Methods

---

getCurrentConsumption(RelativeTime)

Available since RTSJ 2.0

Signature

```
public static
javax.realtime.RelativeTime getCurrentConsumption(RelativeTime
dest)
```

Throws

IllegalStateException when the caller is not a `RealtimeThread`[49].

Returns

The CPU consumption for this release. When `dest` is `null`, return the CPU consumption in an otherwise unused `RelativeTime`[50] instance in the current execution context, otherwise, when `dest` is not `null`, return the CPU consumption in `dest`

getCurrentConsumption

Equivalent to `getCurrentConsumption(null)`.

[49]Section 5.3.2.2
[50]Section 9.4.1.3

Available since RTSJ 2.0

Signature

```
public static
javax.realtime.RelativeTime getCurrentConsumption()
```

getPendingFireCount

This is an accessor method for `fireCount`. The `fireCount` field nominally holds the number of times associated instances of `AsyncEvent`[51] have occurred that have not had the method `handleAsyncEvent()` invoked. It is incremented and decremented by the implementation of the RTSJ. The application logic may manipulate the value in this field for application-specific reasons.

Signature

```
protected
int getPendingFireCount()
```

Returns

The value held by `fireCount`.

getAndClearPendingFireCount

This is an accessor method for `fireCount`. This method atomically sets the value of `fireCount` to zero and returns the value from before it was set to zero. This may used by handlers for which the logic can accommodate multiple releases in a single execution.

The general form for using this is

```
public void handleAsyncEvent()
{
  int numberOfReleases = getAndClearPendingFireCount();
  <handle the events>
}
```

The effect of a call to `getAndClearPendingFireCount` on the scheduling of this AEH depends on the semantics of the scheduler controlling this AEH.

[51] Section 8.4.3.4

Signature

`protected`
`int getAndClearPendingFireCount()`

Returns

The value held by `fireCount` prior to setting the value to zero.

getAndDecrementPendingFireCount

This is an accessor method for `fireCount`. This method atomically decrements, by one, the value of `fireCount` (when it is greater than zero) and returns the value from before the decrement. This method can be used in the `handleAsyncEvent()` method to handle multiple releases:

```
public void handleAsyncEvent()
{
  <setup>
  do
  {
    <handle the event>
  }
  while(getAndDecrementPendingFireCount() > 0);
}
```

This construction is necessary only in the case where a handler wishes to avoid the setup costs since the framework guarantees that `handleAsyncEvent()` will be invoked whenever the `fireCount` is greater than zero. The effect of a call to `getAndDecrementPendingFireCount` on the scheduling of this AEH depends on the semantics of the scheduler controlling this AEH.

Signature

`protected`
`int getAndDecrementPendingFireCount()`

Returns

The value held by `fireCount` prior to decrementing it by one.

getMemoryArea

This is an accessor method for the initial instance of `MemoryArea`[52] associated with `this`.

[52] Section 11.4.3.6

Signature

```
public
javax.realtime.MemoryArea getMemoryArea()
```

Returns

The instance of `MemoryArea`[53] which was passed as the `area` parameter when `this` was created (or the default value when `area` was allowed to default. To determine the current status of the memory area stack associated with `this`, use the static methods defined in the `RealtimeThread`[54] class. That is `RealtimeThread.getCurrentMemoryArea`[55], `RealtimeThread.getInitialMemoryAreaIndex`[56], `RealtimeThread.getMemoryAreaStackDepth`[57].

getMemoryParameters

@inheritDoc

Signature

```
public
javax.realtime.MemoryParameters getMemoryParameters()
```

Returns

@inheritDoc

getReleaseParameters

@inheritDoc

Signature

```
public
javax.realtime.ReleaseParameters getReleaseParameters()
```

Returns

@inheritDoc

[53]Section 11.4.3.6
[54]Section 5.3.2.2
[55]Section 5.3.2.2.2
[56]Section 5.3.2.2.2
[57]Section 5.3.2.2.2

getScheduler

@inheritDoc

Signature

```
public
javax.realtime.Scheduler getScheduler()
```

Returns

@inheritDoc

getSchedulingParameters

@inheritDoc

Signature

```
public
javax.realtime.SchedulingParameters getSchedulingParameters()
```

Returns

@inheritDoc

getProcessingGroupParameters

@inheritDoc

Signature

```
public
javax.realtime.ProcessingGroupParameters
getProcessingGroupParameters()
```

Returns

@inheritDoc

getConfigurationParameters

@inheritDoc

Signature

```
public
javax.realtime.ConfigurationParameters
getConfigurationParameters()
```

Returns

@inheritDoc

setMemoryParameters(MemoryParameters)

@inheritDoc

Signature

```
public
void setMemoryParameters(MemoryParameters memory)
```

Parameters

memory @inheritDoc

Throws

IllegalArgumentException @inheritDoc
IllegalAssignmentError @inheritDoc
IllegalThreadStateException @inheritDoc

setReleaseParameters(ReleaseParameters)

@inheritDoc

Signature

```
public
void setReleaseParameters(ReleaseParameters release)
```

Parameters

release @inheritDoc

Throws

IllegalArgumentException @inheritDoc
IllegalAssignmentError @inheritDoc
IllegalThreadStateException @inheritDoc

setScheduler(Scheduler)

@inheritDoc

Signature

```
public
void setScheduler(Scheduler scheduler)
```

Parameters

scheduler @inheritDoc

Throws

IllegalArgumentException @inheritDoc
IllegalAssignmentError @inheritDoc
SecurityException @inheritDoc
IllegalThreadStateException @inheritDoc

setScheduler(Scheduler, SchedulingParameters, ReleaseParameters, MemoryParameters, ProcessingGroupParameters)

@inheritDoc

Signature

```
public
void setScheduler(Scheduler scheduler, SchedulingParameters
scheduling, ReleaseParameters release, MemoryParameters
memoryParameters, ProcessingGroupParameters group)
```

Parameters

scheduler @inheritDoc
scheduling @inheritDoc
release @inheritDoc
memoryParameters @inheritDoc
group @inheritDoc

Throws

IllegalArgumentException @inheritDoc
IllegalAssignmentError @inheritDoc
IllegalThreadStateException @inheritDoc
SecurityException @inheritDoc

setSchedulingParameters(SchedulingParameters)

@inheritDoc

Signature

```
public
void setSchedulingParameters(SchedulingParameters scheduling)
```

Parameters

scheduling @inheritDoc

Throws

IllegalArgumentException @inheritDoc
IllegalAssignmentError @inheritDoc
IllegalThreadStateException @inheritDoc

setProcessingGroupParameters(ProcessingGroupParameters)

@inheritDoc

Signature

```
public
void setProcessingGroupParameters(ProcessingGroupParameters
group)
```

Parameters

group @inheritDoc

Throws

IllegalArgumentException @inheritDoc
IllegalAssignmentError @inheritDoc
IllegalThreadStateException @inheritDoc

setDaemon(boolean)

Marks this event handler as either a daemon event handler or a user event handler. A realtime virtual machine exits when the only schedulables and threads running are all daemon. This method must be called before the event handler is attached to any event. Once attached, it cannot be changed.
Available since RTSJ 1.0.1

Signature

```
public final
void setDaemon(boolean on)
```

Parameters

on When true, marks this event handler as a daemon handler.

Throws

IllegalThreadStateException when this event handler is attached to an AE.

SecurityException when the current schedulable cannot modify this event handler.

isDaemon

Tests if this event handler is a daemon handler.

Available since RTSJ 1.0.1

Signature

```
public final
boolean isDaemon()
```

Returns

True when this event handler is a daemon handler; false otherwise.

getDispatcher

See Section AsyncTimable.getDispatcher()

Available since RTSJ 2.0

Signature

```
public final
javax.realtime.TimeDispatcher getDispatcher()
```

getMinConsumption(RelativeTime)

Available since RTSJ 2.0

Signature

```
public
javax.realtime.RelativeTime getMinConsumption(RelativeTime dest)
```

getMinConsumption

Available since RTSJ 2.0

Signature

```
public
javax.realtime.RelativeTime getMinConsumption()
```

getMaxConsumption(RelativeTime)

Available since RTSJ 2.0

Signature

```
public
javax.realtime.RelativeTime getMaxConsumption(RelativeTime dest)
```

getMaxConsumption

Available since RTSJ 2.0

Signature

```
public
javax.realtime.RelativeTime getMaxConsumption()
```

wakeUp

Indicate that a sleep has ended.
See Section Schedulable.wakeUp()

Available since RTSJ 2.0

Signature

```
public final
void wakeUp()
```

8.4.3.3 ActiveEventDispatcher

Provides a means of dispatching a set of `ActiveEvent`[58]s. It acts as if it contains a RealtimeThread to perform this task. The priority of this thread can be specified when a dispatcher object is created. The default dispatcher runs at the highest Java realtime priority.

Application code should not extend this class.

Open issue: jjh: should we introduce a Dispatchable type as base type for AsyncTimable and ActiveEvent? The issue arises because TimeDispatcher can dispatch both a Timer which is an ActiveEvent and the sleep in RealtimeThread which is not an ActiveEvent. **End of open issue**

Inheritance

java.lang.Object
  javax.realtime.ActiveEventDispatcher

8.4.3.3.1 Constructors

ActiveEventDispatcher(SchedulingParameters)

Create a new dispatcher.

Signature

```
protected
  ActiveEventDispatcher(SchedulingParameters schedule)
```

[58]Section 8.4.1.1

Parameters

schedule provide scheduling informs to the new object.

8.4.3.3.2 Methods

---

register(T)

Register a POSIX signal with this dispatcher.

Signature

```
public abstract
void register(T event)
throws RegistrationException, IllegalStateException,
IllegalArgumentException
```

Parameters

event to register

Throws

RegistrationException when `event` is already registered.
IllegalStateException when this object has been destroyed.
IllegalArgumentException when `event` is not stopped.

deregister(T)

Deregister a POSIX Signal form this dispatcher. (This is a really naive implementation.)

Signature

```
public abstract
void deregister(T event)
throws DeregistrationException, IllegalStateException,
IllegalArgumentException
```

Parameters

event to deregister

Throws

DeregistrationException when `event` is already registered.
IllegalStateException when this object has been destroyed.

IllegalArgumentException when `event` is not stopped.

destroy

Makes the dispatcher unusable.

Signature

```
public abstract
void destroy()
throws IllegalStateException
```

Throws

IllegalStateException when called on a dispatcher that has one or more registered objects.

8.4.3.4 AsyncEvent

An asynchronous event can have a set of handlers associated with it, and when the event occurs, the `fireCount` of each handler is incremented, and the handlers are released (see `AsyncEventHandler`[59]).

Inheritance

java.lang.Object
  javax.realtime.AbstractAsyncEvent
    javax.realtime.AsyncEvent

8.4.3.4.1 Constructors

AsyncEvent

Create a new `AsyncEvent` object.

Signature

[59]Section 8.4.3.5

```
public
  AsyncEvent()
```

8.4.3.4.2 Methods

fire

When enabled, release the asynchronous events associated with this instance of `AsyncEvent`. When no handlers are attached or this object is disabled the method does nothing, i.e., it skips the release.

- When the instance of `AsyncEvent` has more than one instance of `AsyncEventHandler` with release parameters object of type `AperiodicParameters` attached and the execution of `AsyncEvent.fire()` introduces the requirement to throw at least one type of exception, then all instances of `AsyncEventHandler` not affected by the exception are handled normally
- When the instance of `AsyncEvent` has more than one instance of `AsyncEventHandler` with release parameters object of type `SporadicParameters` attached and the execution of `AsyncEvent.fire()` introduces the simultaneous requirement to throw more than one type of exception or error then `MITViolationException`[60] has precedence over `ArrivalTimeQueueOverflowException`[61].

Signature

```
public
void fire()
```

Throws

MITViolationException Thrown under the base priority scheduler's semantics when there is a handler associated with this event that has its MIT violated by the call to fire (and it has set the minimum inter-arrival time violation behavior to MITViolationExcept). Only the handlers which do not have their MITs violated are released in this situation.

ArrivalTimeQueueOverflowException when the queue of arrival time information overflows. Only the handlers which do not cause this exception to be thrown are released in this situation.

[60] Section 14.2.2.8

[61] Section 14.2.2.1

8.4.3.5 AsyncEventHandler

An asynchronous event handler encapsulates code that is released after an instance of `AsyncEvent`[62] to which it is attached occurs.

It is guaranteed that multiple releases of an event handler will be serialized. It is also guaranteed that (unless the handler explicitly chooses otherwise) for each release of the handler, there will be one execution of the `AsyncEventHandler.handleAsyncEvent()`[63] method. Control over the number of calls to `AsyncEventHandler.handleAsyncEvent()`[64] is given by methods which manipulate a `fireCount`. These may be called by the application via sub-classing and overriding `AsyncEventHandler.handleAsyncEvent()`[65].

Instances of `AsyncEventHandler` with a release parameter of type `SporadicParameters`[66] or `AperiodicParameters`[67] have a list of release times which correspond to the occurrence times of instances of `AsyncEvent`[68] to which they are attached. The minimum interarrival time specified in `SporadicParameters`[69] is enforced when a release time is added to the list. Unless the handler explicitly chooses otherwise, there will be one execution of the code in `AsyncEventHandler.handleAsyncEvent()`[70] for each entry in the list.

The deadline and the time each release event causes the AEH to become eligible for execution are properties of the scheduler that controls the AEH. For the base **scheduler** (at ../../sched_overview-summary.html#AperiodicScheduling), the deadline for each release event is relative to its fire time, and the release takes place at fire time but execution eligibility may be deferred when the queue's MIT violation policy is `SAVE`.

Handlers may do almost anything a realtime thread can do. They may run for a long or short time, and they may block. (Note: blocked handlers may hold system resources.) A handler may not use the `RealtimeThread.waitForNextPeriod()`[71] method.

Normally, handlers are bound to an execution context dynamically when the instances of `AsyncEvent`[72]s to which they are bound occur. This can introduce a

---

[62] Section 8.4.3.4
[63] Section 8.4.3.5.2
[64] Section 8.4.3.5.2
[65] Section 8.4.3.5.2
[66] Section 6.4.2.11
[67] Section 6.4.2.2
[68] Section 8.4.3.4
[69] Section 6.4.2.11
[70] Section 8.4.3.5.2
[71] Section 5.3.2.2.2
[72] Section 8.4.3.4

(small) time penalty. For critical handlers that can not afford the expense, and where this penalty is a problem, `BoundAsyncEventHandler`[73]s can be used.

The scheduler for an asynchronous event handler is inherited from the thread/schedulable that created it. When it was created from a Java thread, the scheduler is the current default scheduler.

The semantics for memory areas that were defined for realtime threads apply in the same way to instances of `AsyncEventHandler` They may inherit a scope stack when they are created, and the single parent rule applies to the use of memory scopes for instances of `AsyncEventHandler` just as it does in realtime threads.

Inheritance

java.lang.Object
 javax.realtime.AbstractAsyncEventHandler
  javax.realtime.AsyncEventHandler

8.4.3.5.1 Constructors

AsyncEventHandler(SchedulingParameters, ReleaseParameters, MemoryParameters, ProcessingGroupParameters, ConfigurationParameters, Runnable)

Create a handler with the given scheduling, release, memory, group, and configuration parameters to run the given logic.
Available since RTSJ 2.0

Signature

```
public
  AsyncEventHandler(SchedulingParameters scheduling, ReleaseParameters release, Memo
```

Parameters

scheduling parameters for scheduling the new handler (and possibly other instances of `Schedulable`[74]). When `scheduling` is `null` and the creator is an

[73] Section 8.4.3.10
[74] Section 6.4.1.2

instance of Schedulable[75], SchedulingParameters[76] is a clone of the creator's value created in the same memory area as `this`. When `scheduling` is `null` and the creator is a Java thread, the contents and type of the new `SchedulingParameters` object is governed by the associated scheduler.

release parameters for scheduling the new handler (and possibly other instances of Schedulable[77]). When `release` is `null` the new `AsyncEventHandler` will use a clone of the default ReleaseParameters[78] for the associated scheduler created in the memory area that contains the `AsyncEventHandler` object.

memory parameters for scheduling the new handler (and possibly other instances of Schedulable[79]). When `memory` is `null`, the new `AsyncEventHandler` receives `null` value for its memory parameters, and the amount or rate of memory allocation for the new handler is unrestricted.

group parameters for providing CPU cost management on a set of Schedulable[80]s. When `null`, `this` will not be associated with any processing group.

initial parameters for reserving space for preallocated exceptions and change implementation specific per Schedulable[81] memory reservations, such as Java stack size, for the new handler (and possibly other instances of Schedulable[82]. When `initial` is `null`, this `AsyncEventHandler` will reserve no space for preallocated exceptions and implementation-specific values will be set to their implementation-defined defaults.

logic The `Runnable` object whose `run()` method will serve as the logic for the new `AsyncEventHandler` . When `logic` is `null`, the `handleAsyncEvent()` method in the new object will serve as its logic.

AsyncEventHandler(SchedulingParameters, ReleaseParameters, Runnable)

Calling this constructor is equivalent to calling `AsyncEventHandler(SchedulingParameters, ReleaseParameters, MemoryParameters, ProcessingGroupParameters, ConfigurationParameters, Runnable)`[83] with arguments `(scheduling, release, null, null, null, logic)`.

[75]Section 6.4.1.2
[76]Section 6.4.2.10
[77]Section 6.4.1.2
[78]Section 6.4.2.8
[79]Section 6.4.1.2
[80]Section 6.4.1.2
[81]Section 6.4.1.2
[82]Section 6.4.1.2
[83]Section 8.4.3.5.1

See Section AsyncEventHandler(SchedulingParameters, ReleaseParameters, MemoryParameters, ProcessingGroupParameters, ConfigurationParameters, Runnable)

Available since RTSJ 2.0

Signature

```
public
  AsyncEventHandler(SchedulingParameters scheduling, ReleaseParameters release, Runn
```

AsyncEventHandler(SchedulingParameters, ReleaseParameters)

Calling this constructor is equivalent to calling `AsyncEventHandler(SchedulingParameters, ReleaseParameters, MemoryParameters, ProcessingGroupParameters, ConfigurationParameters, Runnable)`[84] with arguments `(scheduling, release, null, null, null, null)`
See Section AsyncEventHandler(SchedulingParameters, ReleaseParameters, MemoryParameters, ProcessingGroupParameters, ConfigurationParameters, Runnable)

Available since RTSJ 2.0

Signature

```
public
  AsyncEventHandler(SchedulingParameters scheduling, ReleaseParameters release)
```

AsyncEventHandler(Runnable)

Calling this constructor is equivalent to calling `AsyncEventHandler(SchedulingParameters, ReleaseParameters, MemoryParameters, ProcessingGroupParameters, ConfigurationParameters, Runnable)`[85] with arguments `(null, null, null, null, null, logic)`.

[84] Section 8.4.3.5.1
[85] Section 8.4.3.5.1

Signature

```
public
  AsyncEventHandler(Runnable logic)
```

AsyncEventHandler

Create an instance of `AsyncEventHandler` with default values for all parameters. See Section AsyncEventHandler(SchedulingParameters, ReleaseParameters, MemoryParameters, ProcessingGroupParameters, ConfigurationParameters, Runnable)

Signature

```
public
  AsyncEventHandler()
```

8.4.3.5.2 Methods

handleAsyncEvent

This method holds the logic which is to be executed when any `AsyncEvent`[86] with which this handler is associated is fired. This method will be invoked repeatedly while `fireCount` is greater than zero.

The default implementation of this method invokes the `run` method of any non-null `logic` instance passed to the constructor of this handler.

This AEH acts as a source of "reference" for its initial memory area while it is released.

All throwables from (or propagated through) `handleAsyncEvent` are caught, a stack trace is printed and execution continues as if `handleAsyncEvent` had returned normally.

[86]Section 8.4.3.4

Signature

```
public
void handleAsyncEvent()
```

isPinMemory

Test to see if the default memory area is a pinned scoped memory area.
Available since RTSJ 2.0

Signature

```
public
boolean isPinMemory()
```

Returns

true is the default memory area is a pinned scoped memory area.

run

When used as part of the internal mechanism activated by firing an async event, this method's detailed semantics are defined by the scheduler associated with this handler. The general outline is:

```
enter initial memory area
while (fireCount > 0)
{
  [initiate release]
  fireCount--;
  try { handleAsyncEvent(); }
  catch (Throwable th) { th.printStackTrace(); }
  [effect completion]
}
leave initial memory area
```

All throwables from (or propagated through) `handleAsyncEvent`[87] are caught, a stack trace is printed and execution continues as if `handleAsyncEvent` had returned normally.

[87] Section 8.4.3.5.2

When it is directly invoked, this method invokes `handleAsyncEvent`[88] repeatedly while the `fireCount` is greater than zero; e.g.,

```
while (getAndDecrementPendingFireCount() > 0)
{
  enter initial memory area
  handleAsyncEvent();
  leave initial memory area
}
```

however direct invocation of `run` is not recommended as it may interact with the normal release of this handler.

Applications cannot override this method and thus should use the `logic` parameter at construction, or override `handleAsyncEvent()` in subclasses with the logic of the handler.

Signature

```
public final
void run()
```

8.4.3.6 AsyncLongEvent

A new type of event that carries a long as a payload.
See Section AsyncEvent

Available since RTSJ 2.0

Inheritance

java.lang.Object
  javax.realtime.AbstractAsyncEvent
    javax.realtime.AsyncLongEvent

8.4.3.6.1 Constructors

AsyncLongEvent

[88] Section 8.4.3.5.2

Create a new `AsyncEvent` object.

Signature

```
public
  AsyncLongEvent()
```

8.4.3.6.2 Methods

fire(long)

When enabled, release the asynchronous events associated with this instance of `AsyncLongEvent` with the `long` passed by `fire(long)`[89]. When no handlers are attached or this object is disabled the method does nothing, i.e., it skips the release.

- When the instance of `AsyncLongEvent` is associated with more than one instance of `AsyncLongEventHandler`[90] with release parameters object of type `AperiodicParameters`[91] and the execution of `fire(long)`[92] introduces the requirement to throw at least one type of exception, then all instances of `AsyncLongEventHandler`[93] not affected by the exception are handled normally.
- When this instance of `AsyncLongEvent` is associated with more than one instance of `AsyncLongEventHandler`[94] with release parameters object of type `SporadicParameters`[95] and the execution of `fire(long)`[96] introduces the simultaneous requirement to throw more than one type of exception or error, then `MITViolationException`[97] has precedence over `ArrivalTimeQueueOverflowException`[98].

Signature

[89] Section 8.4.3.6.2
[90] Section 8.4.3.7
[91] Section 6.4.2.2
[92] Section 8.4.3.6.2
[93] Section 8.4.3.7
[94] Section 8.4.3.7
[95] Section 6.4.2.11
[96] Section 8.4.3.6.2
[97] Section 14.2.2.8
[98] Section 14.2.2.1

```
public
void fire(long value)
throws MITViolationException, ArrivalTimeQueueOverflowException
```

Parameters

value is the payload passed to the event.

Throws

MITViolationException Thrown under the base priority scheduler's semantics, when there is a handler associated with this event that has its MIT violated by the call to fire (and it has set the minimum inter-arrival time violation behavior to MITViolationExcept). Only the handlers which do not have their MITs violated are released in this situation.

ArrivalTimeQueueOverflowException when the queue of arrival time information overflows. Only the handlers which do not cause this exception to be thrown are released in this situation.

8.4.3.7 AsyncLongEventHandler

A version of `AbstractAsyncEventHandler`[99] that carries a `long` value as paylaod.
Available since RTSJ 2.0

Inheritance

java.lang.Object
  javax.realtime.AbstractAsyncEventHandler
    javax.realtime.AsyncLongEventHandler

8.4.3.7.1 Constructors

AsyncLongEventHandler(SchedulingParameters, ReleaseParameters, MemoryParameters, ProcessingGroupParameters, ConfigurationParameters, LongConsumer)

Create an asynchronous event handler that receives a `Long` payload with each fire.

[99]Section 8.4.3.2

Signature

```
public
  AsyncLongEventHandler(SchedulingParameters scheduling, ReleaseParameters release,
```

Parameters

scheduling parameters for the new handler.
release parameters for the new handler.
memory parameters for the new handler.
group parameters for the new handler.
sizing parameters for the new handler.
logic is the logic to run for each fire.

AsyncLongEventHandler(SchedulingParameters, ReleaseParameters, LongConsumer)

Calling this constructor is equivalent to calling `AsyncLongEventHandler(SchedulingParameters, ReleaseParameters, MemoryParameters, ProcessingGroupParameters, ConfigurationParameters, LongConsumer)`[100] with arguments `(scheduling, release, null, null, null, logic)`.

Signature

```
public
  AsyncLongEventHandler(SchedulingParameters scheduling, ReleaseParameters release,
```

AsyncLongEventHandler(SchedulingParameters, ReleaseParameters)

Calling this constructor is equivalent to calling `AsyncLongEventHandler(SchedulingParameters, ReleaseParameters, MemoryParameters, ProcessingGroupParameters, ConfigurationParameters, LongConsumer)`[101] with arguments `(scheduling, release, null, null, null, null)`

Signature

[100] Section 8.4.3.7.1
[101] Section 8.4.3.7.1

```
public
  AsyncLongEventHandler(SchedulingParameters scheduling, ReleaseParameters rele
```

AsyncLongEventHandler(LongConsumer)

Calling this constructor is equivalent to calling `AsyncLongEventHandler(SchedulingParameters, ReleaseParameters, MemoryParameters, ProcessingGroupParameters, ConfigurationParameters, LongConsumer)`[102] with arguments `(null, null, null, null, null, null, logic)`.

Signature

```
public
  AsyncLongEventHandler(LongConsumer logic)
```

AsyncLongEventHandler

Create an instance of `AsyncLongEventHandler` with default values for all parameters.
See Section AsyncLongEventHandler(SchedulingParameters, ReleaseParameters, MemoryParameters, ProcessingGroupParameters, ConfigurationParameters, LongConsumer)

Signature

```
public
  AsyncLongEventHandler()
```

8.4.3.7.2 Methods

[102] Section 8.4.3.7.1

handleAsyncEvent(long)

This method holds the logic which is to be executed when any `AsyncEvent`[103] with which this handler is associated is fired. This method will be invoked repeatedly while `fireCount` is greater than zero.

This ALEH is a source of reference for its initial memory area while this ALEH is released.

All throwables from (or propagated through) `handleAsyncEvent` are caught, a stack trace is printed and execution continues as if `handleAsyncEvent` had returned normally.

Signature

```
public
void handleAsyncEvent(long payload)
```

Parameters

payload is the long value associated with a fire.

peekPending

Signature

```
public
long peekPending()
throws IllegalStateException
```

Throws

IllegalStateException when the fire count is zero.

Returns

The long value at the head of the queue of longs to be passed to `handleAsyncEvent(long)`[104].

run

When used as part of the internal mechanism activated by firing an async event, this method's detailed semantics are defined by the scheduler associated with this handler. The general outline is as follows:

```
enter initial memory area
while (fireCount > 0)
```

[103] Section 8.4.3.4

[104] Section 8.4.3.7.2

```
{
  [initiate release]
  fireCount--;
  try
  {
    handleAsyncEvent(value);
  }
  catch (Throwable th)
  {
    th.printStackTrace();
  }
  [effect completion]
}
leave initial memory area
```

All throwables from (or propagated through) `handleAsyncEvent`[105] are caught, a stack trace is printed and execution continues as if `handleAsyncEvent` had returned normally.

When it is directly invoked, this method invokes `handleAsyncEvent`[106] repeatedly while the `fireCount` is greater than zero; e.g.,

```
while (getAndDecrementPendingFireCount() > 0)
enter initial memory area
handleAsyncEvent(value);
leave initial memory area
```

however direct invocation of `run` is not recommended as it may interact with the normal release of this handler.

Applications cannot override this method and thus should use the `logic` parameter at construction, or override `handleAsyncEvent()` in subclasses with the logic of the handler.

Signature

```
public final
void run()
```

8.4.3.8 AsyncObjectEvent

---

[105]Section 8.4.3.7.2

[106]Section 8.4.3.7.2

A new type of event that carries an object as a payload.
See Section AsyncEvent

Available since RTSJ 2.0

Inheritance

java.lang.Object
javax.realtime.AbstractAsyncEvent
javax.realtime.AsyncObjectEvent

8.4.3.8.1 Constructors

AsyncObjectEvent

Create a new `AsyncEvent` object.

Signature

```
public
  AsyncObjectEvent()
```

8.4.3.8.2 Methods

fire(PAYLOAD)

When enabled, fire this instance of `AsyncObjectEvent`. The asynchronous event handlers associated with this event will be released with the object passed by {link `fire(PAYLOAD)`[107]. When no handlers are attached or this object is disabled the method does nothing, i.e., it skips the release.

- When the instance of `AsyncObjectEvent` is associated with more than one instance of `AsyncObjectEventHandler`[108] with release parameters object of type

[107] Section 8.4.3.8.2
[108] Section 8.4.3.9

AperiodicParameters[109] and the execution of fire(PAYLOAD)[110] introduces the requirement to throw at least one type of exception, then all instances of AsyncObjectEventHandler[111] not affected by the exception are handled normally.

- When this instance of AsyncObjectEvent is associated with more than one instance of AsyncObjectEventHandler[112] with release parameters object of type SporadicParameters[113] and the execution of fire(PAYLOAD)[114] introduces the simultaneous requirement to throw more than one type of exception or error, then MITViolationException[115] has precedence over ArrivalTimeQueueOverflowException[116].

Signature

```
public
void fire(PAYLOAD value)
throws MITViolationException, ArrivalTimeQueueOverflowException
```

Parameters

value is the payload passed to the event.

Throws

MITViolationException Thrown under the base priority scheduler's semantics when there is a handler associated with this event that has its MIT violated by the call to fire (and it has set the minimum inter-arrival time violation behavior to MITViolationExcept). Only the handlers which do not have their MITs violated are released in this situation.

ArrivalTimeQueueOverflowException when the queue of arrival time information overflows. Only the handlers which do not cause this exception to be thrown are released in this situation.

8.4.3.9 AsyncObjectEventHandler

A version of AbstractAsyncEventHandler[117] that carries an Object value as pay-

[109]Section 6.4.2.2
[110]Section 8.4.3.8.2
[111]Section 8.4.3.9
[112]Section 8.4.3.9
[113]Section 6.4.2.11
[114]Section 8.4.3.8.2
[115]Section 14.2.2.8
[116]Section 14.2.2.1
[117]Section 8.4.3.2

laod.
Available since RTSJ 2.0

Inheritance

java.lang.Object
javax.realtime.AbstractAsyncEventHandler
javax.realtime.AsyncObjectEventHandler

8.4.3.9.1 Constructors

AsyncObjectEventHandler(SchedulingParameters, ReleaseParameters, MemoryParameters, ProcessingGroupParameters, ConfigurationParameters, java.util.function.Consumer<PAYLOAD>)

Create an asynchronous event handler that receives a `Long` payload with each fire.

Signature

```
public
  AsyncObjectEventHandler(SchedulingParameters scheduling, ReleaseParameters release
```

Parameters

scheduling parameters for the new handler.
release parameters for the new handler.
memory parameters for the new handler.
group parameters for the new handler.
sizing parameters for the new handler.
logic is the logic to run for each fire.

AsyncObjectEventHandler(SchedulingParameters, ReleaseParameters, java.util.function.Consumer<PAYLOAD>)

Calling this constructor is equivalent to calling `AsyncObjectEventHandler(SchedulingParamete ReleaseParameters, MemoryParameters, ProcessingGroupParameters, ConfigurationParameters, Consumer)`[118] with arguments `(scheduling, release, null, null, null, logic)`.

Signature

```
public
  AsyncObjectEventHandler(SchedulingParameters scheduling, ReleaseParameters re
```

AsyncObjectEventHandler(SchedulingParameters, ReleaseParameters)

Calling this constructor is equivalent to calling `AsyncObjectEventHandler(SchedulingParamete ReleaseParameters, MemoryParameters, ProcessingGroupParameters, ConfigurationParameters, Consumer)`[119] with arguments `(scheduling, release, null, null, null, null)`

Signature

```
public
  AsyncObjectEventHandler(SchedulingParameters scheduling, ReleaseParameters re
```

AsyncObjectEventHandler(java.util.function.Consumer<PAYLOAD>)

Calling this constructor is equivalent to calling `AsyncObjectEventHandler(SchedulingParamete ReleaseParameters, MemoryParameters, ProcessingGroupParameters, ConfigurationParameters, Consumer)`[120] with arguments `(null, null, null, null, null, null, logic)`.

Signature

[118] Section 8.4.3.9.1
[119] Section 8.4.3.9.1
[120] Section 8.4.3.9.1

```
public
  AsyncObjectEventHandler(java.util.function.Consumer<PAYLOAD> logic)
```

Parameters

logic is the function to call on the object received.

AsyncObjectEventHandler

Create an instance of `AsyncObjectEventHandler` with default values for all parameters.
See Section AsyncObjectEventHandler(SchedulingParameters, ReleaseParameters, MemoryParameters, ProcessingGroupParameters, ConfigurationParameters, Consumer)

Signature

```
public
  AsyncObjectEventHandler()
```

8.4.3.9.2 Methods

peekPending

Signature

```
public
PAYLOAD peekPending()
throws IllegalStateException
```

Throws

IllegalStateException when the fire count is zero.

Returns

The object reference at the head of the queue of object references to be passed to `handleAsyncEvent(PAYLOAD)`[121]}.

[121] Section ??

run

When used as part of the internal mechanism activated by firing an async event, this method's detailed semantics are defined by the scheduler associated with this handler. The general outline is:

```
enter initial memory area
while (fireCount > 0)
{
  [initiate release]
  fireCount--;
  try { handleAsyncEvent(value); }
  catch (Throwable th) { th.printStackTrace(); }
  [effect completion]
}
leave initial memory area
```

All throwables from (or propagated through) `handleAsyncEvent(PAYLOAD)`[122]} are caught, a stack trace is printed and execution continues as if `handleAsyncEvent` had returned normally.

When it is directly invoked, this method invokes `handleAsyncEvent(PAYLOAD)`[123]} repeatedly while the `fireCount` is greater than zero; e.g.,

```
while (getAndDecrementPendingFireCount() > 0)
{
  enter initial memory area
  handleAsyncEvent(value);
  leave initial memory area
}
```

however direct invocation of `run` is not recommended as it may interact with the normal release of this handler.

Applications cannot override this method and thus should use the `logic` parameter at construction, or override `handleAsyncEvent` in subclasses with the logic of the handler.

Signature

```
public final
void run()
```

[122]Section ??
[123]Section ??

8.4.3.10 BoundAsyncEventHandler

A bound asynchronous event handler is an instance of `AsyncEventHandler`[124] that is permanently bound to a dedicated realtime thread. Bound asynchronous event handlers are for use in situations where the added timeliness is worth the overhead of dedicating an individual realtime thread to the handler. Individual server realtime threads can only be dedicated to a single bound event handler.

Inheritance

java.lang.Object
  javax.realtime.AbstractAsyncEventHandler
    javax.realtime.AsyncEventHandler
      javax.realtime.BoundAsyncEventHandler

Interfaces

BoundAbstractAsyncEventHandler

8.4.3.10.1 Constructors

BoundAsyncEventHandler(SchedulingParameters, ReleaseParameters, MemoryParameters, ProcessingGroupParameters, ConfigurationParameters, Runnable)

Signature

```
public
  BoundAsyncEventHandler(SchedulingParameters scheduling, ReleaseParameters release,
```

BoundAsyncEventHandler(SchedulingParameters, ReleaseParameters, Runnable)

Signature

[124] Section 8.4.3.5

```
public
  BoundAsyncEventHandler(SchedulingParameters scheduling, ReleaseParameters rel
```

BoundAsyncEventHandler(SchedulingParameters, ReleaseParameters)

Signature

```
public
  BoundAsyncEventHandler(SchedulingParameters scheduling, ReleaseParameters rel
```

BoundAsyncEventHandler(Runnable)

Signature

```
public
  BoundAsyncEventHandler(Runnable logic)
```

BoundAsyncEventHandler

Create an instance of `BoundAsyncEventHandler` using default values. This constructor is equivalent to `BoundAsyncEventHandler(null, null, null, null, null, false, null)`

Signature

```
public
  BoundAsyncEventHandler()
```

8.4.3.11 BoundAsyncLongEventHandler

A bound asynchronous event handler is an instance of `AsyncEventHandler`[125] that is permanently bound to a dedicated realtime thread. Bound asynchronous event handlers are for use in situations where the added timeliness is worth the overhead of dedicating an individual realtime thread to the handler. Individual server realtime threads can only be dedicated to a single bound event handler.

Inheritance

java.lang.Object
  javax.realtime.AbstractAsyncEventHandler
    javax.realtime.AsyncLongEventHandler
      javax.realtime.BoundAsyncLongEventHandler

Interfaces

BoundAbstractAsyncEventHandler

8.4.3.11.1 Constructors

BoundAsyncLongEventHandler(SchedulingParameters, ReleaseParameters, MemoryParameters, ProcessingGroupParameters, ConfigurationParameters, LongConsumer)

Signature

```
public
  BoundAsyncLongEventHandler(SchedulingParameters scheduling, ReleaseParameters rele
```

BoundAsyncLongEventHandler(SchedulingParameters, ReleaseParameters, LongConsumer)

Signature

[125] Section 8.4.3.5

```
public
  BoundAsyncLongEventHandler(SchedulingParameters scheduling, ReleaseParameters
```

BoundAsyncLongEventHandler(SchedulingParameters, Release-Parameters)

Signature

```
public
  BoundAsyncLongEventHandler(SchedulingParameters scheduling, ReleaseParameters
```

BoundAsyncLongEventHandler(LongConsumer)

Signature

```
public
  BoundAsyncLongEventHandler(LongConsumer logic)
```

BoundAsyncLongEventHandler

Create an instance of `BoundAsyncEventHandler` using default values. This constructor is equivalent to `BoundAsyncEventHandler(null, null, null, null, null, false, null)`

Signature

```
public
  BoundAsyncLongEventHandler()
```

8.4.3.12 BoundAsyncObjectEventHandler

A bound asynchronous event handler is an instance of `AsyncEventHandler`[126] that is permanently bound to a dedicated realtime thread. Bound asynchronous event handlers are for use in situations where the added timeliness is worth the overhead of dedicating an individual realtime thread to the handler. Individual server realtime threads can only be dedicated to a single bound event handler.

Inheritance

java.lang.Object
  javax.realtime.AbstractAsyncEventHandler
    javax.realtime.AsyncObjectEventHandler
      javax.realtime.BoundAsyncObjectEventHandler

Interfaces

BoundAbstractAsyncEventHandler

8.4.3.12.1 Constructors

BoundAsyncObjectEventHandler(SchedulingParameters, ReleaseParameters, MemoryParameters, ProcessingGroupParameters, ConfigurationParameters, java.util.function.Consumer<Payload>)

Signature

```
public
  BoundAsyncObjectEventHandler(SchedulingParameters scheduling, ReleaseParameters re
```

Parameters

scheduling
release
memory
group
sizing
logic

[126] Section 8.4.3.5

BoundAsyncObjectEventHandler(SchedulingParameters, ReleaseParameters, java.util.function.Consumer<Payload>)

Signature

```
public
  BoundAsyncObjectEventHandler(SchedulingParameters scheduling, ReleaseParamete
```

BoundAsyncObjectEventHandler(SchedulingParameters, ReleaseParameters)

Signature

```
public
  BoundAsyncObjectEventHandler(SchedulingParameters scheduling, ReleaseParamete
```

BoundAsyncObjectEventHandler(java.util.function.Consumer<Payload>

Signature

```
public
  BoundAsyncObjectEventHandler(java.util.function.Consumer<Payload> logic)
```

BoundAsyncObjectEventHandler

Create an instance of `BoundAsyncEventHandler` using default values. This constructor is equivalent to `BoundAsyncEventHandler(null, null, null, null, null, null)`

Signature

```
public
  BoundAsyncObjectEventHandler()
```

8.5 Rationale

The design of the asynchronous event handling facilities was intended to provide the necessary functionality while allowing efficient implementations and catering for a variety of realtime applications. In particular, in some realtime systems there may be a large number of potential events and event handlers (numbering in the thousands or perhaps even the tens of thousands), although at any given time only a small number will be used. Thus it would not be appropriate to dedicate a realtime thread to each event handler. The `RTSJ` addresses this issue by allowing the programmer to specify an event handler either as not bound to a specific realtime thread (the class `AsyncEventHandler`) or alternatively as bound to a dedicated realtime thread (the class `BoundAsyncEventHandler`). The `RTSJ` does not define at what point a non-bound event handler is bound to a realtime thread for its execution. Events are dataless: the fire method does not pass any data to the handler. This was intentional in the interest of simplicity and efficiency. An application that needs to associate data with an `AsyncEvent` can do so explicitly by setting up a buffer; it will then need to deal with buffer overflow issues as required by the application.

The ability to trigger an ATC in a schedulable is necessary in many kinds of realtime applications but must be designed carefully in order to minimize the risks of problems such as data structure corruption and deadlock. There is, invariably, a tension between the desire to cause an ATC to be immediate, and the desire to ensure that certain sections of code are executed to completion.

One basic decision was to allow ATC in a method only if the method explicitly permits this. The default of no ATC is reasonable, since legacy code might be written expecting no ATC, and asynchronously aborting the execution of such a method could lead to unpredictable results. Since the natural way to model ATC is with an exception (`AsynchronouslyInterruptedException`), the way that a method indicates its susceptibility to ATC is by including `AsynchronouslyInterruptedException` in its `throws` clause. Causing this exception to be thrown in a realtime thread `t` as an effect of calling `t.interrupt()` was a natural extension of the semantics of interrupt as currently defined by `java.lang.Thread`.

One ATC-deferred section is `synchronized` code. This is a context that needs to be executed completely in order to ensure a program operates correctly. If `synchronized` code were aborted, a shared object could be left in an inconsistent state. Note that by making synchronized code ATC-deferred, this specification avoids the problems that caused `Thread.stop()` to be deprecated and that have made the use of `Thread.destroy()`, (now also deprecated in Java 1.5) prone to deadlock. If `synchronized` code calls an AI-method and an associated AIE is generated, then if no appropriate handler is present in the synchronized code, the AIE will propagate through the code.

Constructors and `finally` clauses are subject to interruption if the program indicates so. However, if a constructor is aborted, an object might be only partially initialized. If the execution of a `finally` clause in an AI-method is aborted, needed cleanup code might not be performed. Indeed, a `finally` clause in an aborted AI-method will not be executed at all if the abort occurs before its execution begins. It is the programmer's responsibility to ensure that executing these constructs either does not induce unwanted ATC latency (if ATCs are not allowed) or does not produce undesirable results (if ATCs are allowed).

A potential problem with using the exception mechanism to model ATC is that a method with a "catch-all" handler (for example a `catch` clause identifying `Exception` or even `Throwable` as the exception class) can inadvertently intercept an exception intended for a caller. This problem is avoided by having special semantics for catching an AIE. Even though a catch clause may catch an AIE, the exception will be propagated unless the handler invokes the `happened` method from AIE. Thus, if a schedulable is asynchronously interrupted while in a try block that has a handler such as

```
catch (Throwable e)  return;
```

the AIE will remain pending and will be thrown next time control enters or returns to an AI method.

This specification does not provide a special mechanism for terminating a realtime thread; ATC can be used to achieve this effect. This means that, by default, a realtime thread cannot be asynchronously terminated; to support asynchronous termination it needs to enter methods that are AI enabled at frequent intervals. Allowing termination as the default would have been questionable, bringing the same insecurities that are found in `Thread.stop()` and `Thread.destroy()`.

Chapter 9

Time

9.1 Overview

Realtime systems must be able to handle both very short time durations and very long ones. They also needs to distinguish between relative time—a duration of time—and absolute time. Simply using a primitive integral value, such as int or long, does not provide the necessary range. Floating point primitive values, such as float and double, do not provide the necessary precision. Neither provides any type safety. This specification addresses this by requiring three time classes: `HighResolutionTime`, `AbsoluteTime`, and `RelativeTime`, where `HighResolutionTime` is the parent class of the other two.

Instances of `HighResolutionTime` are not created, as the class exists to provide an implementation of the other two classes. An instance of `AbsoluteTime` encapsulates an absolute time. An instance of `RelativeTime` encapsulates a point in time that is relative to some other absolute time value.

All methods returning a time object come in both allocating and nonallocating forms. The classes

- enable describing a point in time with up to nanosecond accuracy and precision (actual accuracy and precision is dependent on the precision of the underlying system),
- enable the distinction between absolute points in time, and times relative to some starting point, and
- provide simple arithmetic operations for using them.

All time handling is based on these classes.

9.2 Definitions

The following terms and abbreviations will be used.

A *time object* is an instance of `AbsoluteTime` or `RelativeTime`.

A *time object* is always associated with a `clock`. By default it is associated with the realtime clock.

The *Epoch* is the standard base time, conventionally January 1 00:00:00 GMT 1970. It is the point from which the realtime clock measures absolute time.

The *time value representation* is a compound format composed of 64 bits of millisecond timing, and 32 bits of nanoseconds within a millisecond. The millisecond constituent uses the 64 bits of a Java `long` while the nanosecond constituent uses the 32 bits of a Java `int`.

The *normalized (canonical) form* for time objects uniquely specifies the values for the millisecond and nanosecond components of a point in time, including the case of 0 milliseconds or 0 nanoseconds, and a negative time value, according to the following three rules:

- When both millisecond and nanosecond components are nonzero they have the same sign. The algebraic time value of the time object is the algebraic sum of the two components.
- The millisecond component represents the algebraic number of milliseconds in the time object, with a range of $[-2^{63}, 2^{63} - 1]$
- The nanosecond component represents the algebraic number of nanoseconds within a millisecond in the time object, that is $[-10^6 + 1, 10^6 - 1]$.

Instances of HighResolutionTime classes always hold a normalized form of a time value. Values that cannot be normalized are not valid; for example, (`MAX_LONG` milliseconds, `MAX_INT` nanoseconds) cannot be normalized and is an illegal value.

The following table has examples of normalized representations.

time in ns	millis	nanos
2000000	2	0
1999999	1	999999
1000001	1	1
1	0	1
0	0	0
-1	0	-1
-999999	0	-999999
-1000000	-1	0
-1000001	-1	-1

9.3 Semantics

This list establishes the semantics that are applicable across the classes of this section. Semantics that apply to particular classes, constructors, methods, and fields will be found in the class description and the constructor, method, and field detail sections.

- All time objects must maintain nanosecond precision and report their values in terms of millisecond and nanosecond constituents.
- Time objects must be constructed from other time objects, from millisecond/nanosecond values, from a `java.util.Date` or obtained as a result of invocations of methods on instances of the `Clock` class.
- Time objects maintain and report time values in normalized form, but the normalized form is not required for input parameter values. This allows computations individually with time constituent parts using the full *signed* range and restrictions of the underlying type.
  - Normalization is accomplished upon method invocation by methods that accept a time object represented with individual component parts, and executed as if the following hold.
    * The nanosecond parameter value, which may be negative, is algebraically added to the scaled millisecond parameter value. The sign of the result provides the sign for any nonzero resulting component.
    * The absolute of the result is then partitioned, giving the number of integral milliseconds for the millisecond component, while the remaining fractional part provides the number of nanoseconds for the nanosecond component.
    * The resulting components are then represented, and reported when necessary, with the above computed sign.
  - Normalization is also performed on the result of operations by methods that perform time object addition and subtraction. Operations are executed using the appropriate arithmetic precision. If the final result of an operation can be represented in normalized form, then the operation must not throw arithmetic exceptions while producing intermediate results.
  - The results of time objects operations and the normalization of results of operations performed with millis and nanos, individually as Java `long` and Java `int` types respectively, are not always equivalent. This is due to the possibility of overflow for nanos values outside of the normalized nanosecond range, that is $[-10^6+1, 10^6-1]$, when performing operations as `int` types, while the same values could be handled with no overflow in time object operations.
  - When invoking setter methods that take as a parameter only one of the two time value components, the other component has implicitly the value of 0.
- Although logically a negative time may represent time before the Epoch or a negative time interval involved in time operations, an `Exception` may be thrown if a negative absolute time or a negative time interval is given as a

parameter to methods. In general, the time values accepted by a method may be a subset of the full time values range, and depend on the method.

- A *time object* is always associated with a `clock`. By default it is associated with the realtime clock. Clocks are involved both in the setting as well as the usage of time objects, for example in comparisons.
- Methods are provided to facilitate the handling of time objects generically via the `HighResolutionTime` class. These methods allow the conversion, according to a `clock`, between `AbsoluteTime` objects and `RelativeTime` objects. These methods also allow the change of `clock` association of a time object. Note that the conversions depend on the time at which they are performed. The semantics of these operations are listed in the following table:

clock association and conversion `this` **has** `clock_a` **and** `ms,ns`	**returned/updated object**
`this_is_absolute.absolute(clock_a)`	`clock_a` `ms,ns`
`this_is_absolute.absolute(clock_b)`	`clock_b` `ms,ns`
`this_is_absolute.absolute(null)`	`realtime_clock` `ms,ns`
`this_is_absolute.relative(clock_a)`	`clock_a` `clock_a.getTime().subtract(ms,ns)`
`this_is_absolute.relative(clock_b)`	`clock_b` `clock_b.getTime().subtract(ms,ns)`
`this_is_absolute.relative(null)`	`realtime_clock` `realtime_clock.getTime().subtract(ms,ns)`
`this_is_relative.relative(clock_a)`	`clock_a` `ms,ns`
`this_is_relative.relative(clock_b)`	`clock_b` `ms,ns`
`this_is_relative.relative(null)`	`realtime_clock` `ms,ns`
`this_is_relative.absolute(clock_a)`	`clock_a` `clock_a.getTime().add(ms,ns)`
`this_is_relative.absolute(clock_b)`	`clock_b` `clock_b.getTime().add(ms,ns)`
`this_is_relative.absolute(null)`	`realtime_clock` `realtime_clock.getTime().add(ms,ns)`

- Time objects must implement the `Comparable` interface if it is available. The `compareTo()` method must be implemented even if the interface is not available.

9.4 Package javax.realtime

9.4.1 Classes

9.4.1.1 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 realtime clock the fixed point is the Epoch (January 1, 1970, 00:00:00 GMT). The correctness of the Epoch as a time base depends on the realtime clock synchronization with an external world time reference. This representation was designed to be compatible with the standard Java representation of an absolute time in the `java.util.Date` class.

A time object in normalized form represents negative time when both components are nonzero and negative, or one is nonzero and negative and the other is zero. For `add` and `subtract` negative values behave as they do in arithmetic.

Caution: This class is explicitly unsafe for multithreading when being changed. Code that mutates instances of this class should synchronize at a higher level.

Inheritance

java.lang.Object
  javax.realtime.HighResolutionTime
    javax.realtime.AbsoluteTime

9.4.1.1.1 Constructors

AbsoluteTime

Equivalent to new `AbsoluteTime(0,0)`.
The clock association is implicitly made with the default realtime clock.

Signature

```
public
  AbsoluteTime()
```

AbsoluteTime(AbsoluteTime)

Make a new `AbsoluteTime` object from the given `AbsoluteTime` object.

The new object will have the same clock association as the `time` parameter.

Signature

```
public
  AbsoluteTime(AbsoluteTime time)

 throws IllegalArgumentException
```

Parameters

time The `AbsoluteTime` object which is the source for the copy.

Throws

IllegalArgumentException when the `time` parameter is `null`.

AbsoluteTime(AbsoluteTime, Chronograph)

Make a new `AbsoluteTime` object from the given `AbsoluteTime` object.

The clock association is made with the `chronograph` parameter. When `chronograph` is `null` the association is made with the default realtime clock.
Available since RTSJ 2.0

Signature

```
public
  AbsoluteTime(AbsoluteTime time, Chronograph chronograph)

 throws IllegalArgumentException
```

Parameters

time The `AbsoluteTime` object which is the source for the copy.
chronograph The chronograph providing the association for the newly constructed object.

Throws

IllegalArgumentException when the `time` parameter is `null`.

AbsoluteTime(Chronograph)

Equivalent to new `AbsoluteTime(0,0,chronograph)`.

The chronograph association is made with the `chronograph` parameter. When `chronograph` is `null` the association is made with the default realtime clock.
Available since RTSJ 2.0

Signature

```
public
  AbsoluteTime(Chronograph chronograph)
```

Parameters

chronograph The chronograph providing the association for the newly constructed object.

AbsoluteTime(BaseCalendar.Date)

Equivalent to new `AbsoluteTime(date.getTime(),0)`. The clock association is implicitly made with the default realtime clock.

Signature

```
public
  AbsoluteTime(BaseCalendar.Date date)

throws IllegalArgumentException
```

Parameters

date The `java.util.Date` representation of the time past the Epoch.

Throws

IllegalArgumentException when the `date` parameter is `null`.

AbsoluteTime(BaseCalendar.Date, Chronograph)

Equivalent to new `AbsoluteTime(date.getTime(),0,chronograph)`.

Warning: While the `date` is used to set the milliseconds component of the new `AbsoluteTime` object (with nanoseconds component set to 0), the new object represents the `date` only when the `chronograph` parameter has an epoch equal to Epoch.

The clock association is made with the `chronograph` parameter. When `chronograph` is `null` the association is made with the default realtime clock.
Available since RTSJ 2.0

Signature

```
public
  AbsoluteTime(BaseCalendar.Date date, Chronograph chronograph)


 throws IllegalArgumentException
```

Parameters

date The `java.util.Date` representation of the time past the Epoch.

chronograph The chronograph providing the association for the newly constructed object.

Throws

IllegalArgumentException when the `date` parameter is `null`.

AbsoluteTime(long, int)

Construct an `AbsoluteTime` object with time millisecond and nanosecond components past the realtime clock's Epoch (00:00:00 GMT on January 1, 1970) based on the parameter `millis` plus the parameter `nanos`. The construction is subject to `millis` and `nanos` parameters normalization. When, after normalization, the time object is negative, the time represented by this is time before the Epoch of its `Chronograph`[1]. The clock association is implicitly made with the default realtime clock.

Signature

```
public
  AbsoluteTime(long millis, int nanos)
```

[1]Section 10.4.1.2

```
throws IllegalArgumentException
```

Parameters

millis is the desired value for the millisecond component of `this`. The actual value is the result of parameter normalization.

nanos is the desired value for the nanosecond component of `this`. The actual value is the result of parameter normalization.

Throws

IllegalArgumentException when there is an overflow in the millisecond component when normalizing.

AbsoluteTime(long, int, Chronograph)

Construct an `AbsoluteTime` object with time millisecond and nanosecond components past the epoch for `chronograph`.

The value of the `AbsoluteTime` instance is based on the parameter `millis` plus the parameter `nanos`. The construction is subject to `millis` and `nanos` parameters normalization. When, after normalization, the time object is negative, the time represented by this is time before `this` chronograph's `epoch`. The chronograph association is made with the `chronograph` parameter. When `chronograph` is `null` the association is made with the default realtime clock.

Note: The start of a chronograph's epoch is an attribute of the chronograph. It is defined as the Epoch (00:00:00 GMT on Jan 1, 1970) for the default realtime clock, but other classes of chronograph may define other epochs.

Available since RTSJ 2.0

Signature

```
public
  AbsoluteTime(long millis, int nanos, Chronograph chronograph)

throws IllegalArgumentException
```

Parameters

millis The desired value for the millisecond component of `this`. The actual value is the result of parameter normalization.

nanos The desired value for the nanosecond component of `this`. The actual value is the result of parameter normalization.

chronograph The chronograph providing the association for the newly constructed object.

Throws

IllegalArgumentException when there is an overflow in the millisecond component when normalizing.

9.4.1.1.2 Methods

absolute(Chronograph)

Return a copy of `this` modified when necessary to have the specified chronograph association. A new object is allocated for the result. This method is the implementation of the `abstract` method of the `HighResolutionTime` base class. No conversion into `AbsoluteTime` is needed in this case. The chronograph association of the result is with the `chronograph` passed as a parameter. When `chronograph` is `null` the association is made with the default realtime clock.

Signature

```
public
javax.realtime.AbsoluteTime absolute(Chronograph chronograph)
```

Parameters

chronograph The `chronograph` parameter is used only as the new chronograph association with the result, since no conversion is needed.

Returns

The copy of `this` in a newly allocated `AbsoluteTime` object, associated with the `chronograph` parameter.

absolute(Chronograph, AbsoluteTime)

Signature

```
public
javax.realtime.AbsoluteTime absolute(Chronograph chronograph,
AbsoluteTime dest)
```

relative(Clock)

Convert the time of `this` to a relative time, using the given instance of `Clock`[2] to determine the current time. The calculation is the current time indicated by the given instance of `Clock`[3] subtracted from the time given by `this`. When `clock` is `null` the default realtime clock is assumed. A destination object is allocated to return the result. The clock association of the result is with the `clock` passed as a parameter.
Deprecated since RTSJ version since version 2.0

Signature

```
public
javax.realtime.RelativeTime relative(Clock clock)
throws ArithmeticException
```

Parameters

clock The instance of `Clock`[4] used to convert the time of `this` into relative time, and the new clock association for the result.

Throws

ArithmeticException when the result does not fit in the normalized format.

Returns

The `RelativeTime` conversion in a newly allocated object, associated with the `clock` parameter.

relative(Chronograph)

Convert the time of `this` to a relative time, using the given instance of `Chronograph`[5] to determine the current time. The calculation is the current time indicated by the given instance of `Chronograph`[6] subtracted from the time given by `this`. When `chronograph` is `null` the default realtime clock is assumed. A destination object is allocated to return the result. The clock association of the result is with the `chronograph` passed as a parameter.
Available since RTSJ 2.0

Signature

[2]Section 10.4.2.1
[3]Section 10.4.2.1
[4]Section 10.4.2.1
[5]Section 10.4.1.2
[6]Section 10.4.1.2

```
public
javax.realtime.RelativeTime relative(Chronograph chronograph)
```

Parameters

chronograph The instance of `Chronograph`[7] used to convert the time of `this` into relative time, and the new clock association for the result.

Throws

ArithmeticException when the result does not fit in the normalized format.

Returns

The `RelativeTime` conversion in a newly allocated object, associated with the `chronograph` parameter.

relative(Chronograph, RelativeTime)

Convert the time of `this` to a relative time, using the given instance of `Chronograph`[8] to determine the current time. The calculation is the current time indicated by the given instance of `Chronograph`[9] subtracted from the time given by `this`. When `chronograph` is `null` the default realtime clock is assumed. When `dest` is not `null`, the result is placed there and returned. Otherwise, a new object is allocated for the result. The clock association of the result is with the `chronograph` passed as a parameter.

Signature

```
public
javax.realtime.RelativeTime relative(Chronograph chronograph,
RelativeTime dest)
```

Parameters

chronograph The instance of `Chronograph`[10] used to convert the time of `this` into relative time, and the new clock association for the result.

dest When `dest` is not `null`, the result is placed there and returned. Otherwise, a new object is allocated for the result.

Throws

ArithmeticException when the result does not fit in the normalized format.

Returns

The `RelativeTime` conversion in `dest` when `dest` is not `null`, otherwise the result is returned in a newly allocated object. It is associated with the `chronograph` parameter.

[7] Section 10.4.1.2
[8] Section 10.4.1.2
[9] Section 10.4.1.2
[10] Section 10.4.1.2

add(long, int)

Create a new object representing the result of adding `millis` and `nanos` to the values from `this` and normalizing the result. The result will have the same clock association as `this`.

Signature

```
public
javax.realtime.AbsoluteTime add(long millis, int nanos)
throws ArithmeticException
```

Parameters

millis The number of milliseconds to be added to `this`.

nanos The number of nanoseconds to be added to `this`.

Throws

ArithmeticException when the result does not fit in the normalized format.

Returns

A new `AbsoluteTime` object whose time is the normalization of `this` plus `millis` and `nanos`.

add(long, int, AbsoluteTime)

Return an object containing the value resulting from adding `millis` and `nanos` to the values from `this` and normalizing the result. When `dest` is not `null`, the result is placed there and returned. Otherwise, a new object is allocated for the result. The result will have the same clock association as `this`, and the clock association with `dest` is ignored.

Signature

```
public
javax.realtime.AbsoluteTime add(long millis, int nanos,
AbsoluteTime dest)
throws ArithmeticException
```

Parameters

millis The number of milliseconds to be added to `this`.

nanos The number of nanoseconds to be added to `this`.

dest When `dest` is not `null`, the result is placed there and returned. Otherwise, a new object is allocated for the result.

Throws

ArithmeticException when the result does not fit in the normalized format.

Returns

the result of the normalization of `this` plus `millis` and `nanos` in `dest` when `dest` is not `null`, otherwise the result is returned in a newly allocated object.

add(RelativeTime)

Create a new instance of `AbsoluteTime` representing the result of adding `time` to the value of `this` and normalizing the result. The `clock` associated with `this` and the `clock` associated with the `time` parameter must be the same, and such association is used for the result.

Signature

```
public
javax.realtime.AbsoluteTime add(RelativeTime time)
throws ArithmeticException, IllegalArgumentException
```

Parameters

time The time to add to `this`.

Throws

IllegalArgumentException when the `clock` associated with `this` and the `clock` associated with the `time` parameter are different, or when the `time` parameter is `null`.

ArithmeticException when the result does not fit in the normalized format.

Returns

A new `AbsoluteTime` object whose time is the normalization of `this` plus the parameter `time`.

add(RelativeTime, AbsoluteTime)

Return an object containing the value resulting from adding `time` to the value of `this` and normalizing the result. When `dest` is not `null`, the result is placed there and returned. Otherwise, a new object is allocated for the result. The `clock` associated with `this` and the `clock` associated with the `time` parameter must be the same, and such association is used for the result. The `clock` associated with the `dest` parameter is ignored.

Signature

```
public
```

```
javax.realtime.AbsoluteTime add(RelativeTime time, AbsoluteTime
dest)
throws ArithmeticException, IllegalArgumentException
```

Parameters

time The time to add to `this`.

dest When `dest` is not `null`, the result is placed there and returned. Otherwise, a new object is allocated for the result.

Throws

IllegalArgumentException when the `clock` associated with `this` and the `clock` associated with the `time` parameter are different, or when the `time` parameter is `null`.

ArithmeticException when the result does not fit in the normalized format.

Returns

the result of the normalization of `this` plus the `RelativeTime` parameter `time` in `dest` when `dest` is not `null`, otherwise the result is returned in a newly allocated object.

getDate

Convert the time given by `this` to a `Date` format. Note that `Date` represents time as milliseconds so the nanoseconds of `this` will be lost.

Signature

```
public
java.util.Date getDate()
throws UnsupportedOperationException
```

Throws

UnsupportedOperationException when the clock associated with `this` does not have the concept of date.

Returns

A newly allocated `Date` object with a value of the time past the Epoch represented by `this`.

set(Date)

Change the time represented by `this` to that given by the parameter. Note that `Date` represents time as milliseconds so the nanoseconds of `this` will be set to 0. The clock association is implicitly made with the default realtime clock.

Signature

```
public
void set(Date date)
throws IllegalArgumentException
```

Parameters

date A reference to a `Date` which will become the time represented by `this` after the completion of this method.

Throws

IllegalArgumentException when the parameter `date` is `null`.

subtract(AbsoluteTime)

Create a new instance of `RelativeTime` representing the result of subtracting `time` from the value of `this` and normalizing the result. The `clock` associated with `this` and the `clock` associated with the `time` parameter must be the same, and such association is used for the result.

Signature

```
public
javax.realtime.RelativeTime subtract(AbsoluteTime time)
throws IllegalArgumentException, ArithmeticException
```

Parameters

time The time to subtract from `this`.

Throws

IllegalArgumentException when the `clock` associated with `this` and the `clock` associated with the `time` parameter are different, or when the `time` parameter is `null`.

ArithmeticException when the result does not fit in the normalized format.

Returns

A new `RelativeTime` object whose time is the normalization of `this` minus the `AbsoluteTime` parameter `time`.

subtract(AbsoluteTime, RelativeTime)

Return an object containing the value resulting from subtracting `time` from the value of `this` and normalizing the result. When `dest` is not `null`, the result is placed there and returned. Otherwise, a new object is allocated for the result. The `clock` associated with `this` and the `clock` associated with the `time` parameter must be the same, and such association is used for the result. The `clock` associated with the `dest` parameter is ignored.

Signature

```
public
javax.realtime.RelativeTime subtract(AbsoluteTime time,
RelativeTime dest)
throws IllegalArgumentException, ArithmeticException
```

Parameters

time The time to subtract from `this`.

dest When `dest` is not `null`, the result is placed there and returned. Otherwise, a new object is allocated for the result.

Throws

IllegalArgumentException when the `clock` associated with `this` and the `clock` associated with the `time` parameter are different, or when the `time` parameter is `null`.

ArithmeticException when the result does not fit in the normalized format.

Returns

the result of the normalization of `this` minus the `AbsoluteTime` parameter `time` in `dest` when `dest` is not `null`, otherwise the result is returned in a newly allocated object.

subtract(RelativeTime)

Create a new instance of `AbsoluteTime` representing the result of subtracting `time` from the value of `this` and normalizing the result. The `clock` associated with `this` and the `clock` associated with the `time` parameter must be the same, and such association is used for the result.

Signature

```
public
javax.realtime.AbsoluteTime subtract(RelativeTime time)
throws IllegalArgumentException, ArithmeticException
```

Parameters

time The time to subtract from `this`.

Throws

IllegalArgumentException when the `clock` associated with `this` and the `clock` associated with the `time` parameter are different, or when the `time` parameter is `null`.
ArithmeticException when the result does not fit in the normalized format.

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

subtract(RelativeTime, AbsoluteTime)

Return an object containing the value resulting from subtracting `time` from the value of `this` and normalizing the result. When `dest` is not `null`, the result is placed there and returned. Otherwise, a new object is allocated for the result. The `clock` associated with `this` and the `clock` associated with the `time` parameter must be the same, and such association is used for the result. The `clock` associated with the `dest` parameter is ignored.

Signature
```
public
javax.realtime.AbsoluteTime subtract(RelativeTime time,
AbsoluteTime dest)
throws IllegalArgumentException, ArithmeticException
```
Parameters
time The time to subtract from `this`.
dest When `dest` is not `null`, the result is placed there and returned. Otherwise, a new object is allocated for the result.

Throws
IllegalArgumentException when the `clock` associated with `this` and the `clock` associated with the `time` parameter are different, or when the `time` parameter is `null`.
ArithmeticException when the result does not fit in the normalized format.

Returns
the result of the normalization of `this` minus the `RelativeTime` parameter `time` in `dest` when `dest` is not `null`, otherwise the result is returned in a newly allocated object.

toString

Create a printable string of the time given by `this`.

The string shall be a decimal representation of the milliseconds and nanosecond values; formatted as follows "(2251 ms, 750000 ns)"

Signature

`public`
`java.lang.String toString()`

Returns

String object converted from the time given by `this`.

9.4.1.2 HighResolutionTime

Class `HighResolutionTime` is the base class for `AbsoluteTime` and `RelativeTime`. It can be used to express time with nanosecond resolution. This class is never used directly; it is abstract and has no public constructor. Instead, one of its subclasses `AbsoluteTime`[11] or `RelativeTime`[12] should be used. When an API is defined that has a `HighResolutionTime` as a parameter, it can take either an absolute or a relative time and will do something appropriate.

Caution: This class is explicitly unsafe for multithreading when being changed. Code that mutates instances of this class should synchronize at a higher level.

Inheritance

java.lang.Object
  javax.realtime.HighResolutionTime

Interfaces

`Comparable`
`Cloneable`

9.4.1.2.1 Methods

---

waitForObject(Object, javax.realtime.HighResolutionTime<T>)

Behaves like `target.wait()` but with the enhancement that it waits with a precision of `HighResolutionTime` and returns true when the associated notify was

[11] Section 9.4.1.1
[12] Section 9.4.1.3

received, false when timeout occured. As for `target.wait()`, there is the possibility of spurious wakeup behavior.

The wait `time` may be relative or absolute, and it is controlled by the `clock` associated with it. When the wait `time` is relative, then the calling thread is blocked waiting on `target` for the amount of time given by `time`, and measured by the associated `clock`. When the wait `time` is absolute, then the calling thread is blocked waiting on `target` until the indicated `time` value is reached by the associated `clock`.
Available since RTSJ 2.0 updated to add a return value.

Signature

```
public static
boolean waitForObject(Object target,
javax.realtime.HighResolutionTime<T> time)
throws InterruptedException, IllegalMonitorStateException,
IllegalArgumentException, UnsupportedOperationException
```

Parameters

T is the type 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. When it is `RelativeTime(0,0)` then wait indefinitely. When it is `null` then wait indefinitely.

Throws

InterruptedException when this schedulable is interrupted by `RealtimeThread.interrupt`[13] or `AsynchronouslyInterruptedException.fire`[14] while it is waiting.
IllegalArgumentException when `time` represents a relative time less than zero.
IllegalMonitorStateException when `target` is not locked by the caller.
UnsupportedOperationException when the wait operation is not supported using the `clock` associated with `time`.

Returns

True when the notify was received before the timeout. False otherwise.

equals(javax.realtime.HighResolutionTime<T>)

Returns `true` when the argument `time` has the same type and values as `this`. Equality includes `clock` association.

[13]Section 5.3.2.2.2
[14]Section 8.4.2.1.2

Signature

```
public
boolean equals(javax.realtime.HighResolutionTime<T> time)
```

Parameters

time Value compared to `this`.

Returns

`true` when the parameter `time` is of the same type and has the same values as `this`.

getClock

Returns a reference to the `clock` associated with `this`.

Available since RTSJ 1.0.1

Signature

```
public
javax.realtime.Clock getClock()
throws UnsupportedOperationException
```

Throws

UnsupportedOperationException when the time is based on a `Chronograph`[15] that is not a `Clock`[16].

Returns

A reference to the `clock` associated with `this`.

getChronograph

Get a reference to the `Chronograph`[17] associated with `this`.

Available since RTSJ 2.0

Signature

```
public
```

[15]Section 10.4.1.2
[16]Section 10.4.2.1
[17]Section 10.4.1.2

```
javax.realtime.Chronograph getChronograph()
```

Returns

A reference to the Chronograph[18] associated with this.

getMilliseconds

Get the milliseconds component of this.

Signature

```
public
long getMilliseconds()
```

Returns

The milliseconds component of the time represented by this.

getNanoseconds

Get the nanoseconds component of this.

Signature

```
public
int getNanoseconds()
```

Returns

The nanoseconds component of the time represented by this.

set(javax.realtime.HighResolutionTime<T>)

Change the value represented by this to that of the given time. The clock associated with this is set to be the clock associated with the time parameter. **Available since RTSJ 1.0.1 The description of the method in 1.0 was erroneous.**

Signature

```
public
void set(javax.realtime.HighResolutionTime<T> time)
```

[18]Section 10.4.1.2

Parameters

time The new value for `this`.

Throws

IllegalArgumentException when the parameter `time` is `null`.
ClassCastException when the type of `this` and the type of the parameter `time` are not the same.

set(long)

Sets the millisecond component of `this` to the given argument, and the nanosecond component of `this` to 0. This method is equivalent to `set(millis, 0)`.

Signature

```
public
void set(long millis)
```

Parameters

millis This value shall be the value of the millisecond component of `this` at the completion of the call.

set(long, int)

Sets the millisecond and nanosecond components of `this`. The setting is subject to parameter normalization. When after normalization the time is negative then the time represented by `this` is set to a negative value, but note that negative times are not supported everywhere. For instance, a negative relative time is an invalid value for a periodic thread's period.

Signature

```
public
void set(long millis, int nanos)
throws IllegalArgumentException
```

Parameters

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.

Throws

IllegalArgumentException when there is an overflow in the millisecond component while normalizing.

hashCode

Returns a hash code for this object in accordance with the general contract of **name**. Time objects that are `equals) equal`[19] have the same hash code.

Signature

```
public
int hashCode()
```

Returns

The hashcode value for this instance.

clone

Return a clone of `this`. This method should behave effectively as when it constructed a new object with the visible values of `this`. The new object is created in the current allocation context.

Available since RTSJ 1.0.1

Signature

```
public
java.lang.Object clone()
```

compareTo(T)

Compares `this HighResolutionTime` with the specified `HighResolutionTime time`.

Signature

```
public
int compareTo(T time)
```

Parameters

[19]Section 9.4.1.2.1

time Compares with the time of `this`.

Throws

ClassCastException when the `time` parameter is not of the same class as `this`.
IllegalArgumentException when the `time` parameter is not associated with the same clock as `this`, or when the `time` parameter is `null`.

Returns

a negative integer, zero, or a positive integer as this object is less than, equal to, or greater than `time`.

equals(Object)

Returns `true` when the argument `object` has the same type and values as `this`. Equality includes `clock` association.

Signature

```
public
boolean equals(Object object)
```

Parameters

object Value compared to `this`.

Returns

`true` when the parameter `object` is of the same type and has the same values as `this`.

absolute(Chronograph)

Convert the time of `this` to an absolute time, using the given instance of `Chronograph`[20] to determine the current time when necessary. When `chronograph` is `null` the realtime clock is assumed.

A destination object is allocated to return the result. The clock association of the result is the `clock` passed as a parameter. See the subclass comments for more specific information.

Signature

```
public abstract
javax.realtime.AbsoluteTime absolute(Chronograph chronograph)
```

Parameters

[20] Section 10.4.1.2

chronograph is the instance of `Chronograph`[21] used to convert the time of `this` into absolute time, and the new chronograph association for the result.

Returns

The `AbsoluteTime` conversion in a newly allocated object, associated with the `clock` parameter.

absolute(Chronograph, AbsoluteTime)

Convert the time of `this` to an absolute time, using the given instance of `Chronograph`[22] to determine the current time when necessary. When `chronograph` is `null` the realtime chronograph is assumed. When `dest` is not `null`, the result is placed there and returned. Otherwise, a new object is allocated for the result. The chronograph association of the result is the `chronograph` passednas a parameter. See the subclass comments for more specific information.

Signature

```
public abstract
javax.realtime.AbsoluteTime absolute(Chronograph chronograph,
AbsoluteTime dest)
```

Parameters

chronograph The instance of `Chronograph`[23] used to convert the time of `this` into absolute time, and the new chronograph association for the result.

dest When `dest` is not `null`, the result is placed there and returned. Otherwise, a new object is allocated for the result.

Returns

The `AbsoluteTime` conversion in `dest` when `dest` is not `null`, otherwise the result is returned in a newly allocated object. It is associated with the `chronograph` parameter.

relative(Chronograph)

Convert the time of `this` to a relative time, using the given instance of `Chronograph`[24] to determine the current time when necessary. When `chronograph` is `null` the realtime chronograph is assumed. A destination object is allocated to return

[21] Section 10.4.1.2
[22] Section 10.4.1.2
[23] Section 10.4.1.2
[24] Section 10.4.1.2

the result. The chronograph association of the result is the `chronograph` passed as a parameter. See the subclass comments for more specific information.

Signature

```
public abstract
javax.realtime.RelativeTime relative(Chronograph chronograph)
```

Parameters

chronograph The instance of `Chronograph`[25] used to convert the time of `this` into relative time, and the new chronograph association for the result.

Returns

The `RelativeTime` conversion in a newly allocated object, associated with the `chronograph` parameter.

relative(Chronograph, RelativeTime)

Convert the time of `this` to a relative time, using the given instance of `Chronograph`[26] to determine the current time when necessary. When `chronograph` is `null` the realtime chronograph is assumed. When `dest` is not `null`, the result is placed there and returned. Otherwise, a new object is allocated for the result. The chronograph association of the result is the `chronograph` passed as a parameter. See the subclass comments for more specific information.

Signature

```
public abstract
javax.realtime.RelativeTime relative(Chronograph chronograph,
RelativeTime dest)
```

Parameters

chronograph The instance of `Chronograph`[27] used to convert the time of `this` into relative time, and the new chronograph association for the result.

dest When `dest` is not `null`, the result is placed there and returned. Otherwise, a new object is allocated for the result.

Returns

The `RelativeTime`[28] conversion in `dest` when `dest` is not `null`, otherwise the result is returned in a newly allocated object.

[25] Section 10.4.1.2
[26] Section 10.4.1.2
[27] Section 10.4.1.2
[28] Section 9.4.1.3

9.4.1.3 RelativeTime

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.

The time interval is kept in normalized form. The range goes from `[(-2^63) milliseconds + (-10^6+1) nanoseconds]` to `[(2^63-1) milliseconds + (10^6-1) nanoseconds]`.

A negative interval relative to now represents time in the past. For `add` and `subtract` negative values behave as they do in arithmetic.

Caution: This class is explicitly unsafe for multithreading when being changed. Code that mutates instances of this class should synchronize at a higher level.

Inheritance

java.lang.Object
  javax.realtime.HighResolutionTime
    javax.realtime.RelativeTime

9.4.1.3.1 Constructors

RelativeTime

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

The clock association is implicitly made with the realtime clock.

Signature

```
public
  RelativeTime()
```

RelativeTime(Chronograph)

Equivalent to `new RelativeTime(0,0,chronograph)`.

The chronograph association is made with the `chronograph` parameter. When `chronograph` is `null` the association is made with the default realtime clock.

Available since RTSJ 2.0

Signature

```
public
  RelativeTime(Chronograph chronograph)
```

Parameters

chronograph The chronograph providing the association for the newly constructed object.

RelativeTime(long, int)

Construct a `RelativeTime` object representing an interval based on the parameter `millis` plus the parameter `nanos`. The construction is subject to `millis` and `nanos` parameters normalization. When there is an overflow in the millisecond component when normalizing then an `IllegalArgumentException` will be thrown. The clock association is implicitly made with the realtime clock.

Signature

```
public
  RelativeTime(long millis, int nanos)


 throws IllegalArgumentException
```

Parameters

millis The desired value for the millisecond component of `this`. The actual value is the result of parameter normalization.

nanos The desired value for the nanosecond component of `this`. The actual value is the result of parameter normalization.

Throws

IllegalArgumentException when there is an overflow in the millisecond component when normalizing.

RelativeTime(long, int, Chronograph)

Construct a `RelativeTime` object representing an interval based on the parameter `millis` plus the parameter `nanos`. The construction is subject to `millis` and `nanos` parameters normalization. When there is an overflow in the millisecond component when normalizing then an `IllegalArgumentException` will be thrown.

The chronograph association is made with the `chronograph` parameter. When `chronograph` is `null` the association is made with the default realtime clock.
Available since RTSJ 2.0

Signature

```
public
  RelativeTime(long millis, int nanos, Chronograph chronograph)


 throws IllegalArgumentException
```

Parameters

millis The desired value for the millisecond component of `this`. The actual value is the result of parameter normalization.
nanos The desired value for the nanosecond component of `this`. The actual value is the result of parameter normalization.
chronograph The chronograph providing the association for the newly constructed object.

Throws

IllegalArgumentException when there is an overflow in the millisecond component when normalizing.

RelativeTime(RelativeTime)

Make a new `RelativeTime` object from the given `RelativeTime` object. The new object will have the same clock association as the `time` parameter.

Signature

```
public
  RelativeTime(RelativeTime time)
```

Parameters

time The `RelativeTime` object which is the source for the copy.

RelativeTime(RelativeTime, Chronograph)

Make a new `RelativeTime` object from the given `RelativeTime` object.

The clock association is made with the `chronograph` parameter. When `chronograph` is `null` the association is made with the realtime clock.

Available since RTSJ 2.0

Signature

```
public
  RelativeTime(RelativeTime time, Chronograph chronograph)


throws IllegalArgumentException
```

Parameters

time The `RelativeTime` object which is the source for the copy.

chronograph The chronograph providing the association for the newly constructed object.

Throws

IllegalArgumentException when the `time` parameter is `null`.

9.4.1.3.2 Methods

---

add(long, int)

Create a new object representing the result of adding `millis` and `nanos` to the values from `this` and normalizing the result. The result will have the same clock association as `this`. An `ArithmeticException` is when the result does not fit in the normalized format.

Signature

```
public
javax.realtime.RelativeTime add(long millis, int nanos)
throws ArithmeticException
```

Parameters

millis The number of milliseconds to be added to `this`.

nanos The number of nanoseconds to be added to `this`.

Throws

ArithmeticException when the result does not fit in the normalized format.

Returns

A new `RelativeTime` object whose time is the normalization of `this` plus `millis` and `nanos`.

add(long, int, RelativeTime)

Return an object containing the value resulting from adding `millis` and `nanos` to the values from `this` and normalizing the result. When `dest` is not `null`, the result is placed there and returned. Otherwise, a new object is allocated for the result. The result will have the same clock association as `this`, and the clock association with `dest` is ignored.

Signature

```
public
javax.realtime.RelativeTime add(long millis, int nanos,
RelativeTime dest)
throws ArithmeticException
```

Parameters

millis The number of milliseconds to be added to `this`.

nanos The number of nanoseconds to be added to `this`.

dest When `dest` is not `null`, the result is placed there and returned. Otherwise, a new object is allocated for the result.

Throws

ArithmeticException when the result does not fit in the normalized format.

Returns

the result of the normalization of `this` plus `millis` and `nanos` in `dest` when `dest` is not `null`, otherwise the result is returned in a newly allocated object.

add(RelativeTime)

Create a new instance of `RelativeTime` representing the result of adding `time` to the value of `this` and normalizing the result.

The `clock` associated with `this` and the `clock` associated with the `time` parameter are expected to be the same, and such association is used for the result.

Signature

```
public
javax.realtime.RelativeTime add(RelativeTime time)
throws IllegalArgumentException, ArithmeticException
```

Parameters

time The time to add to `this`.

Throws

IllegalArgumentException when the `clock` associated with `this` and the `clock` associated with the `time` parameter are different, or when the `time` parameter is `null`.

ArithmeticException when the result does not fit in the normalized format.

Returns

A new `RelativeTime` object whose time is the normalization of `this` plus the parameter `time`.

add(RelativeTime, RelativeTime)

Return an object containing the value resulting from adding `time` to the value of `this` and normalizing the result. When `dest` is not `null`, the result is placed there and returned. Otherwise, a new object is allocated for the result.

The `clock` associated with `this` and the `clock` associated with the `time` parameter are expected to be the same, and such association is used for the result.

The `clock` associated with the `dest` parameter is ignored.

Signature

```
public
javax.realtime.RelativeTime add(RelativeTime time, RelativeTime
dest)
throws IllegalArgumentException, ArithmeticException
```

Parameters

time The time to add to `this`.

dest When `dest` is not `null`, the result is placed there and returned. Otherwise, a new object is allocated for the result.

Throws

IllegalArgumentException when the `clock` associated with `this` and the `clock` associated with the `time` parameter are different, or when the `time` parameter is `null`.

ArithmeticException when the result does not fit in the normalized format.

Returns

the result of the normalization of `this` plus the `RelativeTime` parameter `time` in `dest` when `dest` is not `null`, otherwise the result is returned in a newly allocated object.

subtract(RelativeTime)

Create a new instance of `RelativeTime` representing the result of subtracting `time` from the value of `this` and normalizing the result.

The `clock` associated with `this` and the `clock` associated with the `time` parameter are expected to be the same, and such association is used for the result.

Signature

```
public
javax.realtime.RelativeTime subtract(RelativeTime time)
throws IllegalArgumentException, ArithmeticException
```

Parameters

time The time to subtract from `this`.

Throws

IllegalArgumentException when the `clock` associated with `this` and the `clock` associated with the `time` parameter are different, or when the `time` parameter is `null`.

ArithmeticException when the result does not fit in the normalized format.

Returns

A new `RelativeTime` object whose time is the normalization of `this` minus the parameter `time` parameter `time`.

subtract(RelativeTime, RelativeTime)

Return an object containing the value resulting from subtracting the value of `time` from the value of `this` and normalizing the result. When `dest` is not `null`, the result is placed there and returned. Otherwise, a new object is allocated for the result.

The `clock` associated with `this` and the `clock` associated with the `time` parameter are expected to be the same, and such association is used for the result.

The `clock` associated with the `dest` parameter is ignored.

Signature

```
public
```

```
javax.realtime.RelativeTime subtract(RelativeTime time,
RelativeTime dest)
throws IllegalArgumentException, ArithmeticException
```

Parameters

time The time to subtract from `this`.

dest When `dest` is not `null`, the result is placed there and returned. Otherwise, a new object is allocated for the result.

Throws

IllegalArgumentException when the when the `clock` associated with `this` and the `clock` associated with the `time` parameter are different, or when the `time` parameter is `null`.

ArithmeticException when the result does not fit in the normalized format.

Returns

the result of the normalization of `this` minus the `RelativeTime` parameter `time` in `dest` when `dest` is not `null`, otherwise the result is returned in a newly allocated object.

scale(int)

Change the length of this relative time by multiplying it by `factor`.
Available since RTSJ 2.0

Signature

```
public
javax.realtime.RelativeTime scale(int factor)
```

Parameters

factor by which to increase the time interval.

Returns

a new object with `value` of this scaled by `factor`.

scale(RelativeTime, int)

Set `time` to the value of `this` time by multiplied by `factor`.
Available since RTSJ 2.0

Signature

```
public
javax.realtime.RelativeTime scale(RelativeTime time, int factor)
```

Parameters

time in which to store the results.

factor by which to increase the time interval.

Returns

`time` with the value of `this` scaled by `factor`

compareToZero

Compare `this` to relative time zero returning the result of the comparison. Equivalent to constantZero.compareTo(this)

Available since RTSJ 2.0

Signature

```
public
int compareToZero()
```

Returns

negative when `this` is less than zero, 0, when it is equal to zero and a positive when `this` is greater than zero.

toString

Create a printable string of the time given by `this`.

The string shall be a decimal representation of the milliseconds and nanosecond values; formatted as follows "(2251 ms, 750000 ns)"

Signature

```
public
java.lang.String toString()
```

Returns

String object converted from the time given by `this`.

compareTo(RelativeTime)

Signature

```
public
int compareTo(RelativeTime time)
```

absolute(Chronograph)

Available since RTSJ 2.0

See Section HighResolutionTime.absolute(Chronograph)

Signature

```
public
javax.realtime.AbsoluteTime absolute(Chronograph chronograph)
```

absolute(Chronograph, AbsoluteTime)

Available since RTSJ 2.0

See Section HighResolutionTime.absolute(Chronograph, AbsoluteTime)

Signature

```
public
javax.realtime.AbsoluteTime absolute(Chronograph chronograph,
AbsoluteTime dest)
```

relative(Chronograph)

Available since RTSJ 2.0

See Section HighResolutionTime.relative(Chronograph)

Signature

```
public
javax.realtime.RelativeTime relative(Chronograph chronograph)
```

relative(Chronograph, RelativeTime)

Available since RTSJ 2.0

See Section HighResolutionTime.relative(Chronograph, RelativeTime)

Signature

```
public
javax.realtime.RelativeTime relative(Chronograph chronograph,
RelativeTime dest)
```

9.5 Rationale

Time is the essence of realtime systems, and a method of expressing absolute time with sub-millisecond precision is an absolute minimum requirement. Expressing time in terms of nanoseconds has precedent and allows the implementation to provide time-based services, such as timers, using whatever precision it is capable of while the application requirements are expressed to an arbitrary level of precision.

The standard Java `java.util.Date` class uses milliseconds as its basic unit in order to provide sufficient range for a wide variety of applications. realtime programming generally requires finer resolution, and nanosecond resolution is fine enough for most purposes, but even a 64 bit realtime clock based in nanoseconds would have insufficient range in some situations, so a compound format composed of 64 bits of millisecond timing, and 32 bits of nanoseconds within a millisecond, was chosen.

The expression of millisecond and nanosecond constituents is consistent with other Java interfaces.

The expression of relative times allows for time-based metaphors such as deadline-based periodic scheduling where the cost of the task is expressed as a relative time and deadlines are usually represented as times relative to the beginning of the period.

Chapter 10

Clocks and Timers

10.1 Overview

In order to reason about time, the `RTSJ` needs not only to be able to express times and calculate with them; but it also needs to be able to determine the current time and allow actions to be performed when a given time is reached. For this purpose, the specification defines one interface and four classes: `Chronograph`, `Clock`, `Timer`, `PeriodicTimer`, `and OneShotTimer`.

A chronograph is used to measure time, whereas a clock is used to both measure time and react to its passage: a clock can get the current time and it can trigger timing events. At least one instance of the abstract `Clock` class, which implements `Chronograph`, is provided by the implementation, the system *realtime clock*, and this instance is made available as a singleton. The creation and use of other clocks and chronographs are discussed later (see Section 10.3.2).

The `Timer` classes provide the means of executing code at a particular point in time or repeatedly at a given interval. `Timer` is an abstract class and consequently only its subclasses can be instantiated. The `Timer` class provides the interface and underlying implementation for both one-shot and periodic timers. Instances of `OneShotTimer` and `PeriodicTimer` can be created and rescheduled specifying the initial firing time either as an `AbsoluteTime` or as a `RelativeTime`, to be considered from the application of the start command. The `PhasingPolicy` class defines the relationship between a `PeriodicTimer`'s start time and its first release time when the start time is in the past.

By attaching an `AbstractAsyncEventHandler` to a `Timer`, the program can cause the release of the handler at a given time or after a given interval. An instance of `OneShotTimer` describes an event that is to be triggered at most once (unless restarted after expiration). It may be used as the source for time-outs and watchdog timing. An instance of `PeriodicTimer` fires on a periodic schedule. The period for a `PeriodicTimer` is always specified as a `RelativeTime`.

10.2 Definitions

Understanding the support for clocks and timer requires some basic terms.

A *Timing Mechanism* is something capable of representing and following the progress of time, by means of time values.

A *Chronograph* is a passive timing mechanism, which can only provide the current time.

A *Clock* is an active timing mechanism, which can both provide the current time as well as cause some action when a particular time is reached. All clocks are, by definition, chronographs, but not visa versa.

A *Monotonic Timing Mechanism* is a timing mechanism whose time values always progress in one direction, and

A *Monotonically Increasing* timing mechanism is one whose time values never decrease. Monotonicity is a boolean property, while time synchronization, uniformity, and accuracy are characteristics that depend on agreed tolerances.

Time Synchronization is a relation between two timing mechanisms. Two chronographs are synchronized when the difference between their time values is less than some specified offset. Synchronization in general degrades with time, and may be lost, given a specified offset.

Accuracy is the agreement between a chronograph and the true value that it measures (*e.g.*, absolute wall clock time).

Resolution is the minimal time value interval that can be represented by the clock model.

Precision is the smallest tick size that a particular chronograph will observe.

Uniformity, in this context, refers to the measurement of the progress of time at a consistent rate, with a tolerance on the variability. Uniformity is affected by two other factors, *jitter* and *stability*.

Jitter is caused by short-term and non-cumulative small time variation due to noise sources, such as thermal noise. More practically, jitter refers to the distribution of the differences between when events are actually fired or noticed by the software and when they should have really occurred according to time in the real-world.

Stability in this context refers to temporal stability. Lack of stability accounts for large and often cumulative variations, due to e.g. supply voltage and temperature.

Drift is this rate of this cumulative variation between two timing mechanisms.

Epoch An epoch is the date and time relative to which a computer's clock and timestamp values are determined. The epoch traditionally corresponds to 0 hours, 0 minutes, and 0 seconds (00:00:00) Coordinated Universal Time (UTC) on a specific date.

The Epoch is January 1, 1970, 00:00:00 GMT.

Counting Time is the time accumulated by a `Timer`, while *active*, when created or rescheduled using a `RelativeTime` to specify the initial firing/skipping time. *Counting Time* is zeroed at the beginning of an activation, or when rescheduled, while *active*, before the initial firing/skipping of an activation.

10.3 Semantics

The semantics of chronographs, clocks and timers are not simply functional. Temporal attributes dominate their behavior; therefore, the interaction between classes is critical to the overall understanding of the API. The class descriptions as well as their constructor, method, and field documentation given later provide detailed semantics to support the overall behavior.

10.3.1 Clock Model

Clocks and chronographs are backed by a physical means of measuring time. In practice, each one is driven by an oscillator that has susceptible variation due to its environment. There is always some difference between the desired frequency and the actual frequency of the oscillator, which is a major reason of synchronization loss. The RTSJ Clock model must take this variability into account and therefore establishes several invariants and expectations that can be relied upon by RTSJ applications and in turn must be provided by RTSJ implementations.

- The *resolution* of the RTSJ Clock model is 1 nanosecond. This is the smallest unit of time that can be represented by a chronograph or timer via `HighResolutionTime` and its subclasses.
- The *accuracy* of RTSJ definable chronographs and clocks is outside the scope of this specification. Accuracy is heavily dependent on hardware capabilities and platform characteristics. RTSJ providers and system integrators should characterize accuracy where possible.
- The *precision* of RTSJ definable clock and chronograph (and, by proxy, the precision of the timers associated with cocks) are defined in terms of nanoseconds per observable tick, and provided to the application programmer via the various precision getters on `Clock` and `Chronograph`.
- The realtime clock will be monotonically increasing, and other clocks and chronographs should be monotonically increasing as well.
- Time values returned by a chronograph should not be assumed to be comparable to the time values from another chronograph unless the user has platform-specific knowledge that the chronographs are compatible, except under specific circumstances described below.

- The system or any other realtime clock is not necessarily synchronized with the external world, and the correctness of the epoch as a time base depends on such synchronization. It is as uniform and accurate as allowed by the underlying hardware.

If two Chronograph objects are both referenced to real time and return a value from `getEpochOffset()`, then time values from those Chronographs can be compared by applying their respective corrections. As documented in the `getEpochOffset()` method, its return value represents the offset of the associated Chronograph from the realtime clock Epoch. However, the results of any such comparison must be treated with caution as the accuracy of the two Chronograph objects may be different.

10.3.2 Clocks and Timables

A `Clock` is the basic mechanism of measuring time and triggering events based on the passage of time. A Timer can request a signal from the clock when a given time is reached. That signal should come as closed to the actual time requested as possible. A schedulable also uses a clock to implement the realtime sleep methods. Each clock instance shall be capable of reporting the achievable resolution of timers based on that clock. Each implementation shall have a default clock that is used whenever no other clock is specified. An application can also defined additional clocks.

A `Timer` use a clock to measures time, and informs a `TimeDispatcher` when the time has elapsed (relative time) or has been reached (absolute time). The `TimeDispatcher` causes the release of any `AsyncEventHandler` associated with the `Timer`. In the context of a `Timer`, *triggering* is the action that is performed by a `TimeDispatcher` that informs the `Timer` that it is time to *fire* or *skip*, where skip causes the normal action of fire not to be carried out.

A `Timer` is *active* when it has been started and not stopped since last started and it has a time in the future at which it is expected to fire or skip, else it is *not active*.

In the context of a `Timer`, *enabling* cause the `Timer` to fire when it is triggered, while *disabling* causes the `Timer` to skip when it is triggered. Enabling and disabling act as a mask over firing.

The behavior of a `OneShotTimer` is that of a `Timer` that does not automatically reschedule its triggering after an initial triggering, regardless of whether it fires or skips (when *disabled* and *active* when triggered). It is specified using an initial firing time.

The behavior of a `PeriodicTimer` is that of a `Timer` that automatically reschedules after each triggering, regardless of whether the triggering results in a fire or a skip due to being disabled when triggered. It is specified using an initial firing time and an interval or period used for the self-rescheduling.

A `Clock` can also be used to regulate pauses in execution of any `Schedulable` through a realtime `sleep` method, hence timers and schedulables are classified as timables under the `Timable` interface.

Both `OneShotTimer` and `PeriodTimer` are given an initial firing time. A `PeriodicTimer` receives two clock references, within two `HighResolutionTimer` objects, which must be to the same clock. Thus the specification of the initial firing time and the interval or period must refer to the same clock.

A `Timer` is an `ActiveEvent`. This means that is has an associated dispatcher called `TimeDispatcher`. As with other active events, the application can either use the default dispatcher or create a new one with its own priority and affinity. A schedulable can also have a `TimeDispatcher` to manage sleeping.

Every timable, `Timer` or `Schedulable`, has one clock associated with it, on which the measurement of time will be based. Each clock maintains a list of times, called alarms, that are provided to it from timables. The clock is armed with the next alarm. When that time arrives, the clock signals the `TimeDispatcher` associated with the alarm to signal its timable that the time has arrived.

In the case of a timer, the dispatcher triggers the timer thereby indicating it should fire or skip. In the case of a schedulable, the dispatcher triggers the schedulable to wake up from its sleep. Figure 10.1 illustrates how a timer interacts with a application-defined clock and Figure 10.2 depicts the same for using realtime sleep in a schedulable.

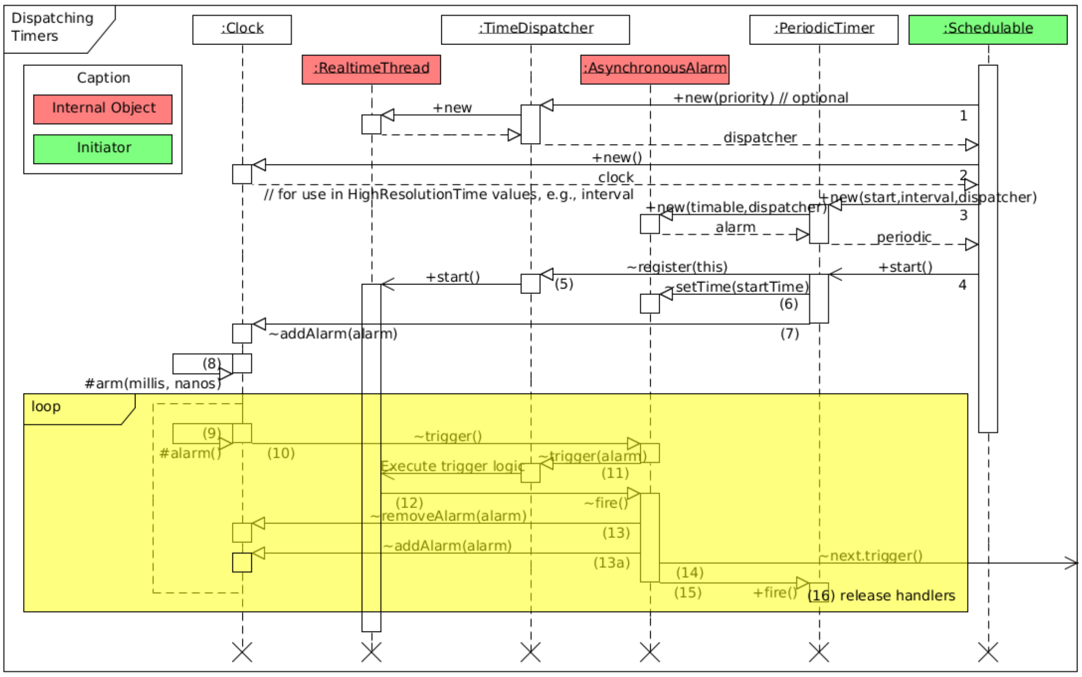
In each case, an external schedulable, depicted on the right, initializes the objects involved. A `TimeDispatcher` and a `Clock` are created. These are used when creating the `Timable` as illustrated with step one and two respectively in both diagrams. A developer can always use a pre-existing clock or dispatcher instead of creating new ones.

Each timable acts as if it had an internal object, depicted as an instance of `Alarm`, to manage the relationship between a timable and its dispatcher and clock. Alarm is shown simply to illustrate this relationship. It is created, step three in both diagrams, when the timable is created and it represents the next alarm that the timable should receive: a fire for a time or a wake up call for a realtime sleep on a schedulable.

At step four, the two sequences diverge. The application start a timer with the start method, but a thread must call a realtime sleep method. In both cases, step four sets the timing in motion.

Steps (5) through (8) set up the time interval. For initiating the trigger for the first time, step (5) registers the timable with its dispatcher. Later starts or sleeps skip this step. Then the time is set in the alarm and the alarm is added to the clock.

Figure 10.1: Sequence Diagram for Using a Timer



When the new alarm is the next alarm to be triggered, the clock arranges to signal that time as in step (8). When the alarm is added anywhere else in the clock queue, step (8) is delayed until the removal of an alarm causes the added alarm to reach the top of the queue.

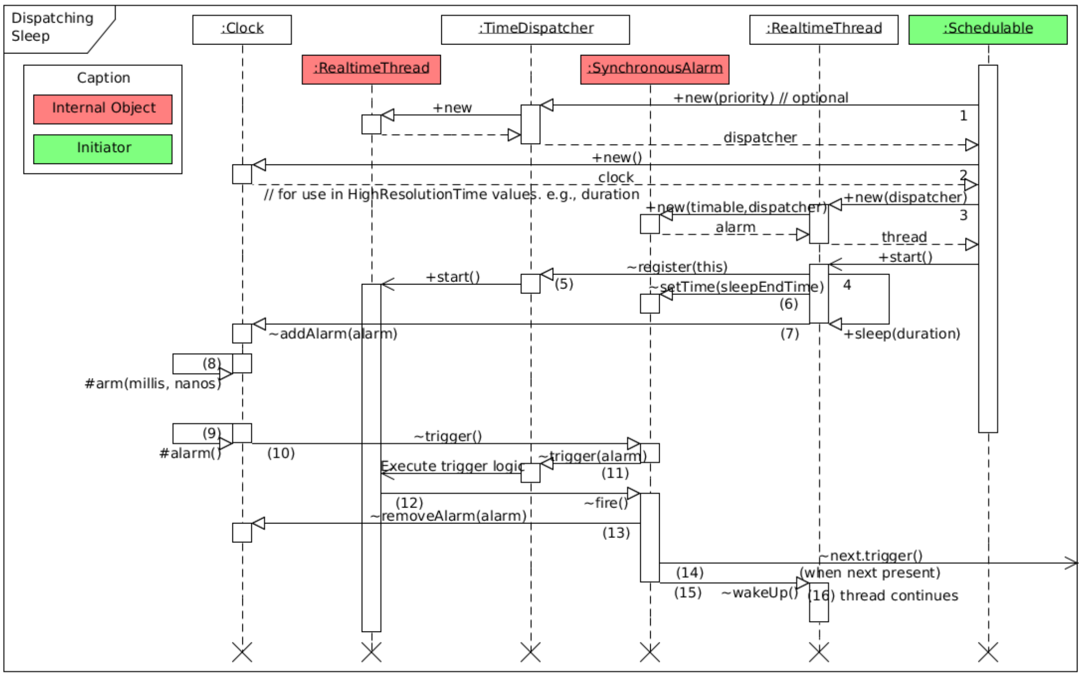
When the alarm time is reached, step (9), the clock triggers the alarm by calling trigger on the alarm event, step (10). This in turn triggers the dispatcher, step (11). This is an asynchronous call that causes the dispatcher's thread to take over control from the clocks interrupt handler.

In step (12), the dispatcher thread removes the alarm from the clock queue, possibly causing a new alarm to become active. In the periodic thread case, the alarm is rescheduled by incrementing the time in the alarm by the interval and adding it back into the queue. In all other cases, no new alarm is set.

In step (13) any subsequent alarms that were scheduled are also kicked off. The Clock queue is a two dimensional queue that is organized by the time of the alarm and, within any given time, the priority order, highest to lowest, of the dispatchers associated with the alarms. The trigger in step (10) always goes to the alarm with the highest priority dispatcher.

Finally in step (14), the dispatcher fires the alarm which results its timable being

Figure 10.2: Sequence Diagram for Realtime Sleep



fired or woken-up. In the case of a timer, this causes all its handlers to be released or, in the case of a schedulable, a sleep being woken up; this is marked as (15) in the diagrams.

`Clocks` and `TimeDispatchers` may be shared among many as timables as the needs of the application dictate. Different dispatchers can be used with a given clock and a dispatcher can service different clocks. The dispatcher should be chosen based on its priority and affinity, whereas a clock should be chosen based on the temporal reference, where the temporal reference may or may not be associated with clock time. For instance, one could use a clock to represent the rotation of a shaft.

10.3.3 Timers

A timer must be associated with a clock. That clock acts as if it provides an interrupt to each of its timers at the next instance of time at which the timer should do something. In other words, a clock fires its timer at a requested time. Timers can be modeled as counters, or as comparators.

10.3.3.1 Counter Model

In the timer model, a timer can be viewed as if every clock interrupt increments a count up to the firing count, initially given by either an instance of `RelativeTime` or computed as the difference between an instance of `AbsoluteTime` and a semantically specified "now" (using the same clock).

- `start` is understood as defining "now" and start counting, `stop` is understood as stop counting. `start` after `stop` may be understood as start counting again from where stopped, or start from scratch after resetting the count.
- In both cases, a delay is introduced.
- An `RTSJ Timer`, when using the counter model, resets the count when it is restarted after being stopped.
- When a `Timer` is created or rescheduled using a `RelativeTime` to specify the initial alarm time, the `RTSJ` keeps the specified initial trigger time as a `RelativeTime` and behaves according to the counter model.

10.3.3.2 Comparator Model

In the comparator model, a Timer can be viewed as if every clock interrupt forces a comparison between an absolute time and a firing time, initially given either as an instance of AbsoluteTime or computed as the sum of an instance of `RelativeTime` and a semantically specified "now" (using the same clock).

- In this model, `start` is understood as start comparing, and possibly the first `start` is understood as defining "now". `stop` is understood as stop comparing. `start` after `stop` may be understood as start comparing again.
- In this case, no delay is introduced.
- When a `Timer` is created or rescheduled using an `AbsoluteTime` to specify the initial triggering time, the `RTSJ` keeps the specified initial firing time as an `AbsoluteTime` and uses the comparator model.

10.3.3.3 Triggering

A clock signals to the associated timable that its alarm time has been reached by triggering the dispatcher associated with the timable. This trigger causes the dispatcher to fire the associated timer. When the timer is active, it releases it handlers and is said to be fired. When the timer is inactive, nothing happens and it is said to be skipped. A stopped timer is never triggered. For this it must be running.

10.3.3.4 Behavior of Timers

There are two kinds of timers defined: `OneShotTimer` and `PeriodicTimer`. As their names imply, the first is used to mark a single time interval and the second is to mark a regularly repeating time interval.

The `OneShotTimer` class shall ensure that each instance is fired at most once at the time specified unless restarted after expiration.

The `PeriodicTimer` class shall enable the period of a timer to be expressed in terms of a `RelativeTime`. The initial firing of a `PeriodicTimer` occurs in response to the invocation of its `start` method, in accordance with the start time passed to its constructor. The `PhasingPolicy` class defines the relationship between the timer's start time and its first firing when the start time is in the past. This initial firing or skipping, may be rescheduled by a call to the `reschedule` method, in accordance with the time passed to that method.

Given an instance of `PeriodicTimer`, let S be the effective time, as an absolute time, at which the initial firing or skipping, of a `PeriodicTimer` is scheduled to occur:

- when the start, or reschedule, time was given as an absolute time, A, and that time is in the future when the timer is made active, then S equals A, otherwise
- when the absolute time has passed when the timer is made active, then S depends on the phasing mode of that instance of `PeriodicTimer`.

The firings of a `PeriodicTimer` are scheduled to occur according to $S + nT$, for $n = 0, 1, 2, ...$ where S is as just specified, and T is the interval of the periodic timer.

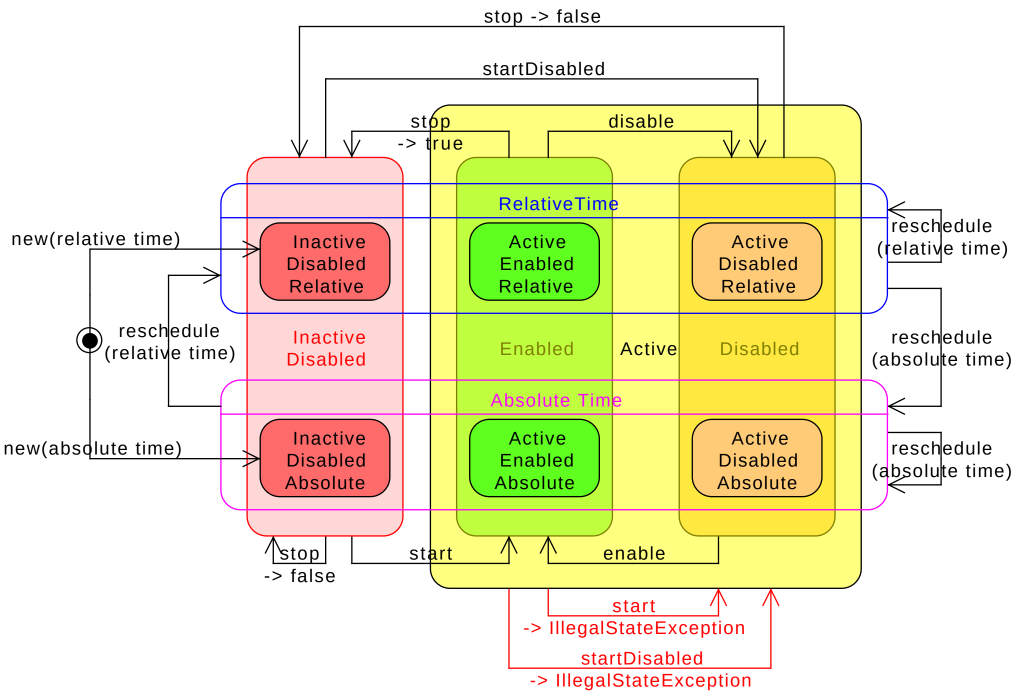
For all timers, when the start or reschedule time is given as a relative time, R, S equals the time at which the *counting time*, started when the timer was made *active*, equals R. The transition to *not-active* by this timer causes the *counting time* to reset, effectively preventing this kind of timer from firing immediately, unless given a time value of 0.

When in a *not-active* state a `Timer` retains the parameters given at construction time or the parameters it had at de-activation time. Those are the parameters that will be used upon invocation of `start` while in that state, unless the parameters are explicitly changed before that, using `reschedule` and `setInterval` as appropriate.

When a `Timer` object is allocated in a scoped memory area, then it will increment the reference count associated with that area. Such a reference count will only be decremented when the `Timer` object is destroyed. (See semantics in the *Memory* chapter for details.) A `Timer` object will not fire before its due time.

The states of a `Timer` are essentially the same as for an `ActiveEvent` as depicted in Figure 8.3. The main difference is that the time used for the next fire may be either an absolute time or a relative time. Figure 10.3 reflects this difference in a UML state diagram.

Figure 10.3: States of a Timer



10.3.3.5 Phasing

Phasing comes into play only when a periodic timer (with period T) starts after its initial time. This can happen when an absolute start time (A) is specified and the start method is called after that time. It is used to determine the effective start time S:

- S is the next multiple of $A + nT$, when phasing is `ADJUST_FORWARD`,
- S is the most recent multiple of $A + nT$, when phasing is `ADJUST_BACKWARD`,
- S is "now," when phasing is `ADJUST_TO_START`, and
- S is undefined and an exception it thrown when phasing is `STRICT_PHASING`.

The default phasing is `ADJUST_TO_START`.

10.4 Package javax.realtime

10.4.1 Interfaces

10.4.1.1 AsyncTimable

A common type for `Timer`[1] and `RealtimeThread`[2] to indicate that they can be associated with a `Clock`[3] and be suspended waiting for time events based on that clock.
Available since RTSJ 2.0

Interfaces
Timable

10.4.1.1.1 Methods

---

fire

Inform the dispatcher associated with this Timeable that a time event has occurred.
Available since RTSJ 2.0

Signature
```
public
void fire()
```

10.4.1.2 Chronograph

---

[1]Section 10.4.2.5
[2]Section 5.3.2.2
[3]Section 10.4.2.1

Available since RTSJ 2.0

10.4.1.2.1 Methods

---

getEpochOffset

Returns the relative time of the offset of the epoch of `this` clock from the Epoch of the realtime clock. The value returned may change over time due to clock drift.

Signature

```
public
javax.realtime.RelativeTime getEpochOffset()
```

Throws

UnsupportedOperationException when the clock does not have the concept of date.

Returns

A newly allocated `RelativeTime`[4] object in the current execution context with the offset past the Epoch for `this` clock. The returned object is associated with `this` clock.

getQueryPrecision

Gets the precision of the time read, the nominal interval between ticks. It is the same as calling `getQueryPrecision(RelativeTime)`[5] with `null` as an argument.

Signature

```
public
javax.realtime.RelativeTime getQueryPrecision()
```

Returns

a value representing the read precision.

---

[4]Section 9.4.1.3
[5]Section 10.4.1.2.1

getQueryPrecision(RelativeTime)

Gets the precision of the time read, the nominal interval between ticks.

Signature

`public`
`javax.realtime.RelativeTime getQueryPrecision(RelativeTime dest)`

Parameters

dest return the relative time value in `dest`. When `dest` is `null`, allocate a new `RelativeTime`[6] instance to hold the returned value.

Returns

`dest` set to values representing the read precision.

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.

Signature

`public`
`javax.realtime.AbsoluteTime getTime()`

Returns

A newly allocated instance of `AbsoluteTime`[7] in the current allocation context, representing the current time. The returned object is associated with `this` clock.

getTime(AbsoluteTime)

Gets the current time in an existing object. The time represented by the given `AbsoluteTime`[8] 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

[6] Section 9.4.1.3
[7] Section 9.4.1.1
[8] Section 9.4.1.1

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.

Signature

```
public
javax.realtime.AbsoluteTime getTime(AbsoluteTime dest)
```

Parameters

dest The instance of `AbsoluteTime`[9] 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. When `dest` is `null`, then nothing happens.

Returns

The instance of `AbsoluteTime`[10] passed as parameter, representing the current time, associated with `this` clock, or `null` when `dest` was `null`.

10.4.1.3 Timable

A type for all classes that can use a `Clock`[11] for timing, either for sleeping or for being released at a given time.

10.4.1.3.1 Methods

getDispatcher

Get the dispatcher associated with this Timeable.

Available since RTSJ 2.0

Signature

```
public
javax.realtime.TimeDispatcher getDispatcher()
```

[9]Section 9.4.1.1
[10]Section 9.4.1.1
[11]Section 10.4.2.1

10.4.2 Classes

10.4.2.1 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.

Note that while all `Clock` implementations use representations of time derived from `HighResolutionTime`, which expresses its time in milliseconds and nanoseconds, a particular `Clock` may track time that is not delimited in seconds or not related to wall clock time in any particular fashion (*e.g.*, revolutions or event detections). In this case, the `Clock`'s timebase should be mapped to milliseconds and nanoseconds in a manner that is computationally appropriate.

Inheritance

java.lang.Object
  javax.realtime.Clock

Interfaces

Chronograph

10.4.2.1.1 Constructors

Clock(boolean)

Constructor for the abstract class.

Signature

```
protected
  Clock(boolean active)
```

Parameters

active to indicate whether or not the clock can be used to drive events.

10.4.2.1.2 Methods

getRealtimeClock

There is always at least one clock object available: the system realtime clock. This is the default `Clock`.

Signature

```
public static
javax.realtime.Clock getRealtimeClock()
```

Returns

The singleton instance of the default `Clock`

getEpochOffset

Returns the relative time of the offset of the epoch of `this` clock from the Epoch of the realtime clock..

Available since RTSJ 1.0.1

Signature

```
public final
javax.realtime.RelativeTime getEpochOffset()
```

Throws

UnsupportedOperationException when the clock does not have the concept of date.

Returns

A newly allocated `RelativeTime`[12] object with the offset past the Epoch for `this` clock. The returned object is associated with `this` clock.

getDrivePrecision

Gets the precision of the clock for driving events, the nominal interval between ticks that can trigger an event. It is the same as calling `getDrivePrecision(RelativeTime)`[13] with `null` as its argument.

Available since RTSJ 2.0

[12]Section 9.4.1.3
[13]Section 10.4.2.1.2

Signature

```
public abstract
javax.realtime.RelativeTime getDrivePrecision()
```

Returns

a value representing the drive precision.

getDrivePrecision(RelativeTime)

Gets the precision of the clock for driving events, the nominal interval between ticks that can trigger an event. The result may be larger than that of `getQueryPrecision(RelativeTime)`[14].

Available since RTSJ 2.0

Signature

```
public abstract
javax.realtime.RelativeTime getDrivePrecision(RelativeTime dest)
```

Parameters

dest return the relative time value in `dest`. When `dest` is `null`, it allocates a new `RelativeTime`[15] instance to hold the returned value.

Returns

`dest` set to values representing the drive precision.

getQueryPrecision

Gets the precision of the time read, the nominal interval between ticks. It is the same as calling `getQueryPrecision(RelativeTime)`[16] with `null` as an argument.

Available since RTSJ 2.0

Signature

```
public abstract
javax.realtime.RelativeTime getQueryPrecision()
```

[14] Section 10.4.2.1.2
[15] Section 9.4.1.3
[16] Section 10.4.2.1.2

Returns

the value representing the read precision.

getQueryPrecision(RelativeTime)

Gets the precision of the time read, the nominal interval between ticks. The result may be smaller than that of `getDrivePrecision(RelativeTime)`[17], when the clock is tied to some system tick for releasing events.

Available since RTSJ 2.0

Signature

```
public abstract
javax.realtime.RelativeTime getQueryPrecision(RelativeTime dest)
```

Parameters

dest return the relative time value in `dest`. When `dest` is `null`, allocate a new `RelativeTime`[18] instance to hold the returned value.

Returns

`dest` set to values representing the read precision.

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.

Signature

```
public final
javax.realtime.AbsoluteTime getTime()
```

Returns

A newly allocated instance of `AbsoluteTime`[19] representing the current time. The returned object is associated with `this` clock.

[17]Section 10.4.2.1.2
[18]Section 9.4.1.3
[19]Section 9.4.1.1

getTime(AbsoluteTime)

Gets the current time in an existing object. The time represented by the given `AbsoluteTime`[20] 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.
Available since RTSJ 1.0.1 The return value is updated from `void` to `AbsoluteTime`.

Signature

```
public abstract
javax.realtime.AbsoluteTime getTime(AbsoluteTime dest)
```

Parameters

dest The instance of `AbsoluteTime`[21] object which will be updated in place. The clock association of the `dest` parameter is set to `this`. When `dest` is not `null` the returned object is associated with `this` clock. When `dest` is `null`, a new object is allocated.

Returns

The instance of `AbsoluteTime`[22] passed as parameter, representing the current time, associated with `this` clock, or `null` when `dest` was `null`.

setResolution(RelativeTime)

Set the resolution of `this`. For some hardware clocks setting resolution is impossible and when this method is called on those clocks, then an `UnsupportedOperationException` is thrown.

Signature

```
public abstract
void setResolution(RelativeTime resolution)
```

Parameters

resolution The new resolution of `this`, when the requested value is supported by `this` clock. When `resolution` is smaller than the minimum resolution

[20] Section 9.4.1.1
[21] Section 9.4.1.1
[22] Section 9.4.1.1

supported by `this` clock then it throws `IllegalArgumentException`. When the requested `resolution` is not available and it is larger than the minimum resolution, then the clock will be set to the closest resolution that the clock supports, via truncation. The value of the `resolution` parameter is not altered. The clock association of the `resolution` parameter is ignored.

Throws

IllegalArgumentException when `resolution` is `null`, or when the requested `resolution` is smaller than the minimum resolution supported by this clock.

UnsupportedOperationException when the clock does not support setting its resolution.

triggerAlarm

Code in the abstract base `Clock` is called by a subclass to signal that the time of the next alarm has been reached. It will trigger a `TimeDispatcher`[23], which in turn will cause a fire on an associated `AsyncTimable`[24]

For timers that do not drive events, this should simply do nothing.

This method should be implemented with a runtime complexity not exceeding O(1). Implementations exceeding this bound shall explicitly document the complexity their implementation. **Available since RTSJ 2.0**

Signature

```
protected final
void triggerAlarm()
```

setAlarm(long, int)

Implemented by subclasses to set the time for the next alarm. When there is an alarm outstanding when called, the subclass must override the old time. This should never be called from application or library code. It is intended to be called only from the `javax.realtime` package.
Available since RTSJ 2.0

Signature

[23]Section 10.4.2.4
[24]Section 10.4.1.1

```
protected abstract
void setAlarm(long milliseconds, int nanoseconds)
```

Parameters

milliseconds of the next alarm.

nanoseconds of the next alarm.

clearAlarm

Implemented by subclasses to cancel the current outstanding alarm.

Available since RTSJ 2.0

Signature

```
protected abstract
void clearAlarm()
```

10.4.2.2 OneShotTimer

A timed `AsyncEvent`[25] that is driven by a `Clock`[26]. It will fire once, when the clock time reaches the time-out time, unless restarted after expiration. When the timer is *disabled* at the expiration of the indicated time, the firing is lost (*skipped*). After expiration, the `OneShotTimer` becomes *not-active* and *disabled*. When the clock time has already passed the time-out time, it will fire immediately after it is started or after it is rescheduled while *active*.

Semantics details are described in the `Timer`[27] pseudocode and compact graphic representation of state transitions.

Caution: This class is explicitly unsafe for multithreading when being changed. Code that mutates instances of this class should synchronize at a higher level.

Inheritance

java.lang.Object
  javax.realtime.AbstractAsyncEvent
    javax.realtime.AsyncEvent
      javax.realtime.Timer
        javax.realtime.OneShotTimer

[25] Section 8.4.3.4
[26] Section 10.4.2.1
[27] Section 10.4.2.5

10.4.2.2.1 Constructors

OneShotTimer(javax.realtime.HighResolutionTime<?>, AsyncEventHandler, TimeDispatcher)

Create an instance of `OneShotTimer`[28], based on the `Clock`[29] associated with the `time` parameter, that will execute its `fire` method according to the given time.
Available since RTSJ 2.0

Signature

```
public
  OneShotTimer(javax.realtime.HighResolutionTime<?> time, AsyncEventHandler handler,

 throws IllegalArgumentException, UnsupportedOperationException,
 IllegalAssignmentError
```

Parameters

time The time used to determine when to fire the event. A `time` value of `null` is equivalent to a `RelativeTime` of 0, and in this case the `Timer` fires immediately upon a call to `start()`.
handler The `AsyncEventHandler`[30] that will be released when `fire` is invoked. When `null`, no handler is associated with this `Timer` and nothing will happen when this event fires unless a handler is subsequently associated with the timer using the `addHandler()` or `setHandler()` method.
dispatcher The `TimeDispatcher`[31] to use for triggering this event.

Throws

IllegalArgumentException when `time` is a `RelativeTime` instance less than zero.
UnsupportedOperationException when the `Chronograph`[32] associated with `time` is not a `Clock`[33].

---

[28] Section 10.4.2.2
[29] Section 10.4.2.1
[30] Section 8.4.3.5
[31] Section 10.4.2.4
[32] Section 10.4.1.2
[33] Section 10.4.2.1

IllegalAssignmentError when this `OneShotTimer` cannot hold a reference to `handler`.

OneShotTimer(javax.realtime.HighResolutionTime<?>, AsyncEventHandler)

The equivalent of calling `OneShotTimer(HighResolutionTime<?>, AsyncEventHandler, TimeDispatcher)`[34] with arguments `time, handler, null`.

Signature

```
public
  OneShotTimer(javax.realtime.HighResolutionTime<?> time, AsyncEventHandler han
```

Parameters

time is the time to release its handlers.
handler is the hanndler to release.

10.4.2.3 PeriodicTimer

An `AsyncEvent`[35] whose `fire` method is executed periodically according to the given parameters. The clock associated with the `Timer` start time must be identical to the the clock associated with the `Timer interval`

The first firing is at the beginning of the first interval.

When an interval greater than 0 is given, the timer will fire periodically. When an interval of 0 is given, the `PeriodicTimer` will only fire once, unless restarted after expiration, behaving like a `OneShotTimer`. In all cases, when the timer is *disabled* when the firing time is reached, that particular firing is lost (*skipped*). When *enabled* at a later time, it will fire at its next scheduled time.

When the clock time has already passed the beginning of the first period, the `PeriodicTimer` will first fire according to the `PhasingPolicy`[36].

Semantics details are described in the `Timer`[37] pseudo-code and compact graphic representation of state transitions.

[34] Section **??**
[35] Section 8.4.3.4
[36] Section 5.3.1.1
[37] Section 10.4.2.5

Caution: This class is explicitly unsafe for multithreading when being changed. Code that mutates instances of this class should synchronize at a higher level.

Inheritance

java.lang.Object
  javax.realtime.AbstractAsyncEvent
    javax.realtime.AsyncEvent
      javax.realtime.Timer
        javax.realtime.PeriodicTimer

10.4.2.3.1 Constructors

PeriodicTimer(javax.realtime.HighResolutionTime<?>, RelativeTime, AsyncEventHandler)

Create a timer that executes its fire method periodically. Equivalent to `PeriodicTimer(start, interval, handler, null)`.

Signature

```
public
  PeriodicTimer(javax.realtime.HighResolutionTime<?> start, RelativeTime interval, A

 throws IllegalArgumentException, IllegalAssignmentError
```

PeriodicTimer(javax.realtime.HighResolutionTime<?>, RelativeTime, AsyncEventHandler, TimeDispatcher)

Create a timer that executes its fire method periodically.
Available since RTSJ 2.0

Signature

```
public
  PeriodicTimer(javax.realtime.HighResolutionTime<?> start, RelativeTime interval, A
```

`throws IllegalArgumentException, IllegalAssignmentError, UnsupportedOperationException`

Parameters

start The time that specifies when the first interval begins, based on the clock associated with it. The first firing of the timer is modified according the `PhasingPolicy` when the timer is started. A `start` value of `null` is equivalent to a `RelativeTime` of 0.

interval The period of the timer. Its usage is based on the clock specified by the `clock` parameter. When `interval` is zero or `null`, the period is ignored and the firing behavior of the `PeriodicTimer` is that of a `OneShotTimer`[38].

handler The `AsyncEventHandler`[39] that will be released when `fire` is invoked. When `null`, no handler is associated with this `Timer` and nothing will happen when this event fires unless a handler is subsequently associated with the timer using the `addHandler()` or `setHandler()` method.

dispatcher is the dispatcher to use for triggering this event.

Throws

IllegalArgumentException when `start` or `interval` is a `RelativeTime` instance with a value less than zero; or the clocks associated with `start` and `interval` are not the identical.

IllegalAssignmentError when this `PeriodicTimer` cannot hold references to `handler`, `clock` and `interval`.

UnsupportedOperationException when the `Chronograph`[40] associated with `time` is not a `Clock`[41].

10.4.2.3.2 Methods

---

addHandler(AsyncEventHandler)

Add a handler to the set of handlers associated with this event. It overrides the method in `AbstractAsyncEvent`[42] to allow the use of handlers with `PeriodicParameters`[43], but these parameters must match the period of this timer, otherwise `IllegalArgumentException` is thrown.

[38] Section 10.4.2.2
[39] Section 8.4.3.5
[40] Section 10.4.1.2
[41] Section 10.4.2.1
[42] Section 8.4.3.1
[43] Section 6.4.2.4

Available since RTSJ 2.0

Signature

```
public
void addHandler(AsyncEventHandler handler)
throws IllegalArgumentException, IllegalAssignmentError
```

Parameters

handler a new handler to add to the list of handlers already associated with `this`. When `handler` is already associated with the event, the call has no effect.

Throws

IllegalArgumentException when `handler` is `null` or the handler has `PeriodicParameters`[44] with a period that does not match the period of `this`.

IllegalAssignmentError when this `AsyncEvent` cannot hold a reference to `handler`.

setHandler(AsyncEventHandler)

Associate a new handler with this event and remove all existing handlers. It overrides the method in `AbstractAsyncEvent`[45] to allow the use of handlers with `PeriodicParameters`[46], but these parameters must match the period of this timer, otherwise `IllegalArgumentException` is thrown.

Available since RTSJ 2.0

Signature

```
public
void setHandler(AsyncEventHandler handler)
throws IllegalArgumentException, IllegalAssignmentError
```

Parameters

handler The instance of `AbstractAsyncEventHandler`[47] to be associated with `this`. When `handler` is `null`, no handler will be associated with `this`, i.e., behave effectively as when `setHandler(null)` invokes `removeHandler(AbstractAsyncEventHandler)` for each associated handler.

[44] Section 6.4.2.4
[45] Section 8.4.3.1
[46] Section 6.4.2.4
[47] Section 8.4.3.2

Throws

IllegalArgumentException when `handler` has `PeriodicParameters`[48] with a period that does not match the period of `this`.

IllegalAssignmentError when this `AsyncEvent` cannot hold a reference to `handler`.

start(PhasingPolicy)

Start the timer with the specified `PhasingPolicy`[49].

Available since RTSJ 2.0

Signature

```
public
void start(PhasingPolicy phasingPolicy)
throws LateStartException, IllegalArgumentException
```

Parameters

phasingPolicy determines what happens when the start is too late.

Throws

LateStartException when this method is called after its absolute start time and the `phasingPolicy` is `PhasingPolicy.STRICT_PHASING`[50].

IllegalArgumentException when the start time of this timer is not an absolute time, or `phasingPolicy` is `null`.

start(boolean, PhasingPolicy)

Start the timer with the specified `PhasingPolicy`[51] and the specified disabled state.

Available since RTSJ 2.0

Signature

```
public
void start(boolean disabled, PhasingPolicy phasingPolicy)
```

[48]Section 6.4.2.4
[49]Section 5.3.1.1
[50]Section 5.3.1.1.1
[51]Section 5.3.1.1

`throws LateStartException, IllegalArgumentException`

Parameters

disabled deternine the mode of start: `true` for enabled and `false` for disabled for consistency with `Timer.start(boolean)`[52].

phasingPolicy determines what happens when the start is too late.

Throws

LateStartException when this method is called after its absolute start time and the `phasingPolicy` is `PhasingPolicy.STRICT_PHASING`[53].

IllegalArgumentException when the start time of this timer is not an absolute time, or `phasingPolicy` is `null`.

getClock

Each instance can only be associated with a single clock, which this method can obtain.

Available since RTSJ 1.0.1

Signature

```
public
javax.realtime.Clock getClock()
throws IllegalStateException
```

Throws

IllegalStateException when `this` has been destroyed.

Returns

the instance of `Clock`[54] that is associated with `this`.

createReleaseParameters

Create a release parameters object with new objects containing copies of the values corresponding to this timer. When the `PeriodicTimer` interval is greater than 0, create a `PeriodicParameters`[55] object with a start time and period that correspond to the next firing (or skipping) time, and interval, of this timer. When

[52]Section 10.4.2.5.4
[53]Section 5.3.1.1.1
[54]Section 10.4.2.1
[55]Section 6.4.2.4

the interval is 0, create an `AperiodicParameters`[56] object, since in this case the timer behaves like a `OneShotTimer`[57].

When this timer is active, then the start time is the next firing (or skipping) time returned as an `AbsoluteTime`[58]. Otherwise, the start time is the initial firing (or skipping) time, as set by the last call to `Timer:reschedulereschedule`[59], or when there was no such call, by the constructor of this timer.

Signature

```
public
javax.realtime.ReleaseParameters createReleaseParameters()
```

Throws

IllegalStateException when this `Timer` has been *destroyed*.

Returns

A new release parameters object with new objects containing copies of the values corresponding to this timer. When the interval is greater than zero, return a new instance of `PeriodicParameters`[60]. When the interval is zero return a new instance of `AperiodicParameters`[61].

getFireTime

Get the time at which this `PeriodicTimer` is next expected to fire or to skip. When the `PeriodicTimer` is *disabled*, the returned time is that of the skipping of the firing. When the `PeriodicTimer` is *not-active* it throws `IllegalStateException`.

Signature

```
public
javax.realtime.AbsoluteTime getFireTime()
throws ArithmeticException, IllegalStateException
```

Throws

ArithmeticException when the result does not fit in the normalized format.

IllegalStateException when this `Timer` has been *destroyed*, or when it is *not-active*.

Returns

[56] Section 6.4.2.2
[57] Section 10.4.2.2
[58] Section 9.4.1.1
[59] Section 10.4.2.5.4
[60] Section 6.4.2.4
[61] Section 6.4.2.2

The absolute time at which `this` is next expected to fire or to skip, in a newly allocated `AbsoluteTime`[62] object. When the timer has been created or re-scheduled (see `Timer.reschedule(HighResolutionTime time)`[63]) using an instance of `RelativeTime` for its time parameter then it will return the sum of the current time and the `RelativeTime` remaining time before the timer is expected to fire/skip. Within a periodic timer activation, the returned time is associated with the start clock before the first fire (or skip) time, and associated with the interval clock otherwise.

getFireTime(AbsoluteTime)

Get the time at which this `PeriodicTimer` is next expected to fire or to skip. When the `PeriodicTimer` is *disabled*, the returned time is that of the skipping of the firing. When the `PeriodicTimer` is *not-active* it throws `IllegalStateException`.
Available since RTSJ 1.0.1

Signature

```
public
javax.realtime.AbsoluteTime getFireTime(AbsoluteTime dest)
```

Parameters

dest The instance of `AbsoluteTime`[64] which will be updated in place and returned. The clock association of the `dest` parameter is ignored. When `dest` is `null` a new object is allocated for the result.

Throws

ArithmeticException when the result does not fit in the normalized format.

IllegalStateException when this `Timer` has been *destroyed*, or when it is *not-active*.

Returns

The instance of `AbsoluteTime`[65] passed as parameter, with time values representing the absolute time at which `this` is expected to fire or to skip. When the `dest` parameter is `null` the result is returned in a newly allocated object. When the timer has been created or re-scheduled (see `Timer.reschedule(HighResolutionTime time)`[66]) using an instance of `RelativeTime` for its time parameter then it will

[62]Section 9.4.1.1
[63]Section 10.4.2.5.4
[64]Section 9.4.1.1
[65]Section 9.4.1.1
[66]Section 10.4.2.5.4

return the sum of the current time and the `RelativeTime` remaining time before the timer is expected to fire/skip. Within a periodic timer activation, the returned time is associated with the start clock before the first fire (or skip) time, and associated with the interval clock otherwise.

getInterval

Gets the interval of `this Timer`.

Signature

```
public
javax.realtime.RelativeTime getInterval()
```

Throws

IllegalStateException when this `Timer` has been *destroyed*.

Returns

The `RelativeTime` instance assigned as this periodic timer's interval by the constructor or `setInterval(RelativeTime)`[67].

setInterval(RelativeTime)

Reset the `interval` value of `this`.

Signature

```
public
void setInterval(RelativeTime interval)
```

Parameters

interval A `RelativeTime`[68] object which is the interval used to reset this Timer. A `null interval` is interpreted as `RelativeTime(0,0)`.
The `interval` does not affect the first firing (or skipping) of a timer's activation. At each firing (or skipping), the next fire (or skip) time of an *active* periodic timer is established based on the `interval` currently in use. Resetting the `interval` of an *active* periodic timer only effects future fire (or skip) times after the next.

Throws

[67]Section 10.4.2.3.2
[68]Section 9.4.1.3

IllegalArgumentException when `interval` is a `RelativeTime` instance with a value less than zero, or the clock associated with `interval` is different to the clock associated with `this`.
IllegalAssignmentError when this `PeriodicTimer` cannot hold a reference to `interval`.
IllegalStateException when this `Timer` has been *destroyed*.

10.4.2.4 TimeDispatcher

A dispatcher for time events: `Timer`[69] and `RealtimeThread.sleep`[70].
Available since RTSJ 2.0

Inheritance

java.lang.Object
  javax.realtime.ActiveEventDispatcher
    javax.realtime.TimeDispatcher

10.4.2.4.1 Constructors

TimeDispatcher(SchedulingParameters)

Create a new dispatcher, whose dispatching thread runs with the given scheduling parameters.

Signature

```
public
  TimeDispatcher(SchedulingParameters schedule)
```

Parameters

schedule give the parameters for scheduling this dispatcher

[69] Section 10.4.2.5
[70] Section 5.3.2.2.2

10.4.2.4.2 Methods

register(Timable)

Register a `AsyncTimable`[71] with this dispatcher.

Signature

```
public
void register(Timable target)
throws RegistrationException, IllegalStateException,
IllegalArgumentException
```

Parameters

target to register

Throws

RegistrationException when `target` is already registered.
IllegalStateException when this object has been destroyed.
IllegalArgumentException when `target` is not stopped.

deregister(Timable)

Deregister a `AsyncTimable`[72] from this dispatcher.

Signature

```
public
void deregister(Timable target)
throws DeregistrationException, IllegalStateException,
IllegalArgumentException
```

Parameters

target to deregister

Throws

DeregistrationException when `target` is not already registered.
IllegalStateException when this object has been destroyed.
IllegalArgumentException when `target` is not stopped.

[71] Section 10.4.1.1
[72] Section 10.4.1.1

destroy

Release all resources thereby making the dispatcher unusable.

Signature

```
public
void destroy()
throws IllegalStateException
```

Throws

IllegalStateException when called on a dispatcher that has one or more registered `AsyncTimable`[73] objects.

10.4.2.5 Timer

A *timer* is a timed event that measures time according to a given `Clock`[74]. This class defines basic functionality available to all timers. Applications will generally use either `PeriodicTimer`[75] to create an event that is fired repeatedly at regular intervals, or `OneShotTimer`[76] for an event that just fires once at a specific time. A timer is always associated with at least one `Clock`[77], which provides the basic facilities of something that ticks along following some time line (realtime, CPU-time, user-time, simulation-time, etc.). All timers are created *disabled* and do nothing until `start()` is called.

10.4.2.5.1 Pseudo-Code Representation of State Transitions for Timer

An implementation shall behave effectively as if it implemented the following pseudo-code. Only absolute and relative time behaviors are shown as rational time has been deprecated.

NOTE: The pseudo-code does not take into account any issue of synchronization, it just shows the functionality, and the intended behavior is obtained with groups of and'ed statements interpreted as atomic. This is relevant, for example, in cases where the *firing* of an `AsyncEventHandler`[78] is part of the statements preceding a state transition. While the *firing* causes the release of the handler before the state

[73] Section 10.4.1.1
[74] Section 10.4.2.1
[75] Section 10.4.2.3
[76] Section 10.4.2.2
[77] Section 10.4.2.1
[78] Section 8.4.3.5

transition, the execution of the handler does not take place until after the state transition has completed.

The pseudo-code is a model, it should be interpreted as running continuously, with instructions that take no time.

```
absolute construction state is {not-active, disabled, absolute}
with nextTargetTime = absoluteTime
last_rescheduled_with_AbsoluteTime = TRUE
[(if PeriodicTimer) period = interval]

relative construction state is {not-active, disabled, relative}
with nextDurationTime = relativeTime
last_rescheduled_with_AbsoluteTime = FALSE
[(if PeriodicTimer) period = interval]

{not-active, disabled, absolute}
  [(if PeriodicTimer)
   set released_or_skipped_in_current_activation = FALSE]
  enable -> no state change, do nothing
  disable -> no state change, do nothing
  stop -> no state change, return FALSE
  start ->
    [if last_rescheduled_with_AbsoluteTime
     then
       [set targetTime = nextTargetTime
        [if targetTime < currentTime
         then set targetTime = currentTime]
        then go to state {active, enabled, absolute}]
     else
       [set countingTime = 0
        and set durationTime = nextDurationTime
        then go to state {active, enabled, relative}]]
  isRunning -> return FALSE
  reschedule ->
    [if using an instance of AbsoluteTime
     then
       [reset the nextTargetTime to absoluteTime arg
        and set last_rescheduled_with_AbsoluteTime = TRUE
        and no state change]
     else
```

```
        [reset the nextDurationTime to relativeTime arg
         and set last_rescheduled_with_AbsoluteTime = FALSE
         and go to state {not-active, disabled, relative}]]
  getFireTime -> throws IllegalStateException
  destroy -> go to state {destroyed}
  startDisabled ->
    [if last_rescheduled_with_AbsoluteTime
     then
        [set targetTime = nextTargetTime
         [if targetTime < currentTime
          then set targetTime = currentTime]
         then go to state {active, disabled, absolute}]
     else
        [set countingTime = 0
         and set durationTime = nextDurationTime
         then go to state {active, disabled, relative}]]

{not-active, disabled, relative}
  [(if PeriodicTimer)
   set released_or_skipped_in_current_activation = FALSE]
  enable -> no state change, do nothing
  disable -> no state change, do nothing
  stop -> no state change, return FALSE
  start ->
    [if last_rescheduled_with_AbsoluteTime
     then
        [set targetTime = nextTargetTime
         [if targetTime < currentTime
          then set targetTime = currentTime]
         then go to state {active, enabled, absolute}]
     else
        [set countingTime = 0
         and set durationTime = nextDurationTime
         then go to state {active, enabled, relative}]]
  isRunning -> return FALSE
  reschedule ->
    [if using an instance of AbsoluteTime
     then
        [reset the nextTargetTime to absoluteTime arg
         and set last_rescheduled_with_AbsoluteTime = TRUE
```

```
        and go to state {not-active, disabled, absolute}]
    else
       [reset the nextDurationTime to relativeTime arg
        and set last_rescheduled_with_AbsoluteTime = FALSE
        and no state change]]
  getFireTime -> throws IllegalStateException
  destroy -> go to state {destroyed}
  startDisabled ->
    [if last_rescheduled_with_AbsoluteTime
     then
       [set targetTime = nextTargetTime
        [if targetTime < currentTime
         then set targetTime = currentTime]
        then go to state {active, disabled, absolute}]
     else
       [set countingTime = 0
        and set durationTime = nextDurationTime
        then go to state {active, disabled, relative}]]

{active, enabled, absolute}
  [if currentTime >= targetTime
   then
     [if PeriodicTimer
      then
        [if period > 0
         then
           [fire
            and set released_or_skipped_in_current_activation = TRUE
            and self reschedule
                via targetTime = (targetTime + period)
            and re-enter current state]
         else
           [fire
            and go to state {not-active, disabled, absolute}]]
      else
        [it is a OneShotTimer so
         fire
         and go to state {not-active, disabled, absolute}]]]
  enable -> no state change, do nothing
  disable -> go to state {active, disabled, absolute}
```

```
  stop -> [go to state {not-active, disabled, absolute}
          and return TRUE]
  start -> throws IllegalStateException
  isRunning -> return TRUE
  reschedule ->
    [if NOT released_or_skipped_in_current_activation
     then
       [if using an instance of AbsoluteTime
        then
          [reset the targetTime to absoluteTime arg
           and re-enter current state]
        else
          [reset the durationTime to relativeTime arg
           and set countingTime = 0
           and go to state {active, enabled, relative}]]
     else
       [if using an instance of AbsoluteTime
        then
          [reset the nextTargetTime to absoluteTime arg
           and set last_rescheduled_with_AbsoluteTime = TRUE
           and no state change]
        else
          [reset the nextDurationTime to relativeTime arg
           and set last_rescheduled_with_AbsoluteTime = FALSE
           and no state change]]]
  getFireTime -> return targetTime
  destroy -> go to state {destroyed}
  startDisabled -> throws IllegalStateException

{active, enabled, relative}
  [if countingTime >= durationTime
   then
     [if PeriodicTimer
      then
        [if period > 0
         then
           [fire
            and set released_or_skipped_in_current_activation = TRUE
            and self reschedule
                via durationTime = (durationTime + period)
```

```
            and re-enter current state]
         else
           [fire
            and go to state {not-active, disabled, relative}]]
      else
        [it is a OneShotTimer so
         fire
         and go to state {not-active, disabled, relative}]]]
enable -> no state change, do nothing
disable -> go to state {active, disabled, relative}
stop -> [go to state {not-active, disabled, relative}
         and return TRUE]
start -> throws IllegalStateException
isRunning -> return TRUE
reschedule ->
  [if NOT released_or_skipped_in_current_activation
   then
     [if using an instance of AbsoluteTime
      then
        [reset the targetTime to absoluteTime arg
         and go to state {active, enabled, absolute}]
      else
        [reset the durationTime to relativeTime arg
         and set countingTime = 0
         and re-enter current state]]
   else
     [if using an instance of AbsoluteTime
      then
        [reset the nextTargetTime to absoluteTime arg
         and set last_rescheduled_with_AbsoluteTime = TRUE
         and no state change]
      else
        [reset the nextDurationTime to relativeTime arg
         and set last_rescheduled_with_AbsoluteTime = FALSE
         and no state change]]]
getFireTime ->
  return (currentTime + durationTime - countingTime)
destroy -> go to state {destroyed}
startDisabled -> throws IllegalStateException
```

```
{active, disabled, absolute}
  [if currentTime >= targetTime
   then
     [if PeriodicTimer
      then
        [if period > 0
         then
           [skip
            and set released_or_skipped_in_current_activation = TRUE
            and self reschedule
                via targetTime = (targetTime + period)
            and re-enter current state]
         else
           [skip
            and go to state {not-active, disabled, absolute}]]
      else
        [it is a OneShotTimer so
         skip
         and go to state {not-active, disabled, absolute}]]]
  enable -> go to state {active, enabled, absolute}
  disable -> no state change, do nothing
  stop -> [go to state {not-active, disabled, absolute}
           and return TRUE]
  start -> throws IllegalStateException
  isRunning -> return FALSE
  reschedule ->
    [if NOT released_or_skipped_in_current_activation
     then
       [if using an instance of AbsoluteTime
        then
          [reset the targetTime to absoluteTime arg
           and re-enter current state]
        else
          [reset the durationTime to relativeTime arg
           and set countingTime = 0
           and go to state {active, disabled, relative}]]
     else
       [if using an instance of AbsoluteTime
        then
          [reset the nextTargetTime to absoluteTime arg
```

```
            and set last_rescheduled_with_AbsoluteTime = TRUE
            and no state change]
         else
           [reset the nextDurationTime to relativeTime arg
            and set last_rescheduled_with_AbsoluteTime = FALSE
            and no state change]]]
  getFireTime -> return targetTime
  destroy -> go to state {destroyed}
  startDisabled -> throws IllegalStateException

{active, disabled, relative}
  [if countingTime >= durationTime
   then
     [if PeriodicTimer
      then
        [if period > 0
         then
           [skip
            and set released_or_skipped_in_current_activation = TRUE
            and self reschedule
                via durationTime = (durationTime + period)
            and re-enter current state]
         else
           [skip
            and go to state {not-active, disabled, relative}]]
      else
        [it is a OneShotTimer so
         skip
         and go to state {not-active, disabled, relative}]]]
  enable -> go to state {active, enabled, relative}
  disable -> no state change, do nothing
  stop -> [go to state {not-active, disabled, relative}
           and return TRUE]
  start -> throws IllegalStateException
  isRunning -> return FALSE
  reschedule ->
    [if NOT released_or_skipped_in_current_activation
     then
       [if using an instance of AbsoluteTime
        then
```

[reset the targetTime to absoluteTime arg
and go to state {**active, disabled, absolute**}]
else
[reset the durationTime to relativeTime arg
and set countingTime = 0
and **re-enter current state**]]
else
[if using an instance of AbsoluteTime
then
[reset the nextTargetTime to absoluteTime arg
and set last_rescheduled_with_AbsoluteTime = TRUE
and no state change]
else
[reset the nextDurationTime to relativeTime arg
and set last_rescheduled_with_AbsoluteTime = FALSE
and no state change]]]
getFireTime ->
return (currentTime + durationTime - countingTime)
destroy -> go to state {**destroyed**}
startDisabled -> **throws IllegalStateException**

{**destroyed**}
enable — disable — stop — start — isRunning
— reschedule — getFireTime — destroy
— startDisabled -> **throws IllegalStateException**

The following two methods, without loss of generality and to avoid clutter, have been omitted from the above Pseudo-code.

Every state but {**destroyed**} has:
[(if PeriodicTimer) setInterval -> reset period = interval]
[(if PeriodicTimer) getInterval -> return period]

The state {**destroyed**} has:
[(if PeriodicTimer) setInterval -> throws **IllegalStateException**]
[(if PeriodicTimer) getInterval -> throws **IllegalStateException**]

10.4.2.5.2 Compact Graphic Representation of State Transitions for Timer

The following compact graphic representation, while not as detailed, complements the State Transitions for Timer pseudo-code:

(see image at doc-files/timers_state_machine.gif)

Inheritance

java.lang.Object
javax.realtime.AbstractAsyncEvent
javax.realtime.AsyncEvent
javax.realtime.Timer

Interfaces

AsyncTimable
ActiveEvent

10.4.2.5.3 Constructors

Timer(javax.realtime.HighResolutionTime<?>, AsyncEventHandler, TimeDispatcher)

Create a timer that fires according to the given `time`, based on the `Clock`[79] `clock` and is handled by the specified `AsyncEventHandler`[80] `handler`.
Available since version 2.0

Signature

```
protected
  Timer(javax.realtime.HighResolutionTime<?> time, AsyncEventHandler handler, T

throws IllegalArgumentException, UnsupportedOperationException,
IllegalAssignmentError
```

Parameters

time The time used to determine when to fire the event. A `time` value of `null` is equivalent to a `RelativeTime` of 0, and in this case the `Timer` fires immediately upon a call to `start()`.

[79]Section 10.4.2.1
[80]Section 8.4.3.5

handler The default `handler` to use for this event. When `null`, no `handler` is associated with the timer and nothing will happen when this event fires unless a `handler` is subsequently associated with the timer using the `addHandler()` or `setHandler()` method.

dispatcher The `dispatcher` used to interface between this `timer` and its associated clock. When `null`, the system default dispatcher is used.

Throws

IllegalArgumentException when `time` is a negative `RelativeTime` value.

UnsupportedOperationException when `time` has a `Chronograph`[81] is not a `clock`.

IllegalAssignmentError when this `Timer` cannot hold references to `handler` and `clock`.

10.4.2.5.4 Methods

---

getClock

Return the instance of `Clock`[82] on which this `timer` is based.

Signature

```
public
javax.realtime.Clock getClock()
throws IllegalStateException
```

Throws

IllegalStateException when this `Timer` has been *destroyed*.

Returns

The instance of `Clock`[83] associated with this `Timer`.

getStartTime

Get the start time of this `Timer`. Note that the start time uses copy semantics, so changes made to the value returned by this method do not effect the start time of this `Timer`.

[81]Section 10.4.1.2
[82]Section 10.4.2.1
[83]Section 10.4.2.1

Signature

```
public
javax.realtime.HighResolutionTime<?> getStartTime()
```

Returns

a reference to the time (or start) parameter used when constructing this `Timer`. Since RTSJ 2.0

getEffectiveStartTime

Return a newly-created time representing the time the timer actually started, or when the timer has been rescheduled, the effective start time after the reschedule.
Available since RTSJ 2.0

Signature

```
public
javax.realtime.AbsoluteTime getEffectiveStartTime()
throws IllegalStateException, ArithmeticException
```

Throws

IllegalStateException when the timer is not active or has been destroyed.
ArithmeticException when the result does not fit in the normalized format.

Returns

the time `this` actually started.

getEffectiveStartTime(AbsoluteTime)

Update `dest` to represent the time the timer actually started, or when the timer has been rescheduled, the effective start time after the reschedule. When `dest` is `null`, behave as if `getEffectiveStartTime()`[84] had been called.
Available since RTSJ 2.0

Signature

```
public
javax.realtime.AbsoluteTime getEffectiveStartTime(AbsoluteTime
dest)
throws IllegalStateException, ArithmeticException
```

[84] Section 10.4.2.5.4

Parameters

dest a place to store the time `this` actually started.

Throws

IllegalStateException when the timer is not active or has been destroyed.

ArithmeticException when the result does not fit in the normalized format.

Returns

The time the timer actually started, or when it has been rescheduled, the effective start time after the reschedule.

getLastReleaseTime

Get the last release time of this timer.

Available since RTSJ 2.0

Signature

```
public final
javax.realtime.AbsoluteTime getLastReleaseTime()
```

Throws

IllegalStateException when this timer has not been released since it was last started.

Returns

a reference to a newly-created `AbsoluteTime`[85] object representing this timer's last release time. When the timer has not been released since it was last started, throw an exception.

getLastReleaseTime(AbsoluteTime)

Signature

```
public
javax.realtime.AbsoluteTime getLastReleaseTime(AbsoluteTime
dest)
```

Returns

When `dest` is `null`, return a reference to a newly-created `AbsoluteTime`[86] object representing this timer's last release time. When `dest` is non-null, set

[85] Section 9.4.1.1

[86] Section 9.4.1.1

dest to this timer's last release time. When the timer has not been released, return null. Since RTSJ 2.0

getFireTime

Get the time at which this Timer is expected to fire. When the Timer is *disabled*, the returned time is that of the skipping of the firing. When the Timer is *not-active* it throws IllegalStateException.

Signature

```
public
javax.realtime.AbsoluteTime getFireTime()
throws IllegalStateException, ArithmeticException
```

Throws

ArithmeticException when the result does not fit in the normalized format.

IllegalStateException when this Timer has been *destroyed*, or when it is *not-active*.

Returns

The absolute time at which this is expected to fire (release handlers or skip), in a newly allocated AbsoluteTime[87] object. When the timer has been created or re-scheduled (see Timer.reschedule(HighResolutionTime)[88]) using an instance of RelativeTime for its time parameter then it will return the sum of the current time and the RelativeTime remaining time before the timer is expected to fire/skip. The clock association of the returned time is the clock on which this timer is based.

getFireTime(AbsoluteTime)

Get the time at which this Timer is expected to fire. When the Timer is *disabled*, the returned time is that of the skipping of the firing. When the Timer is *not-active* it throws IllegalStateException.
Available since RTSJ 1.0.1

Signature

```
public
```

[87] Section 9.4.1.1
[88] Section 10.4.2.5.4

```
javax.realtime.AbsoluteTime getFireTime(AbsoluteTime dest)
throws IllegalStateException, ArithmeticException
```

Parameters

dest The instance of AbsoluteTime[89] which will be updated in place and returned. The clock association of the `dest` parameter is ignored. When `dest` is `null` a new object is allocated for the result.

Throws

ArithmeticException when the result does not fit in the normalized format.

IllegalStateException when this `Timer` has been *destroyed*, or when it is *not-active*.

Returns

The instance of AbsoluteTime[90] passed as parameter, with time values representing the absolute time at which `this` is expected to fire (release its handlers or skip). When the `dest` parameter is `null` the result is returned in a newly allocated object. When the timer has been created or re-scheduled (see Timer.reschedule(HighResolutionTime)[91]) using an instance of `RelativeTime` for its time parameter then it will return the sum of the current time and the `RelativeTime` remaining time before the timer is expected to fire. The clock association of the returned time is the clock on which `this` timer is based.

getDispatcher

Signature

```
public
javax.realtime.TimeDispatcher getDispatcher()
```

isActive

Determine the activation state of this happening, i.e., it has been started.

Signature

```
public
boolean isActive()
```

[89]Section 9.4.1.1
[90]Section 9.4.1.1
[91]Section 10.4.2.5.4

Returns

`true` when active, `false` otherwise.

isRunning

Determines if `this` is *active* and is *enabled* such that when the given time occurs it will fire the event. Given the `Timer` current state it answer the question *"Is firing expected?"*.

Signature

```
public
boolean isRunning()
throws IllegalStateException
```

Throws

IllegalStateException when this `Timer` has been *destroyed.*

Returns

`true` when the timer is *active* and *enabled*; oyherwise `false`, when the timer has either not been *started*, it has been *started* but it is *disabled*, or it has been *started* and is now *stopped.*

handledBy(AsyncEventHandler)

Available since RTSJ 1.0.1

Signature

```
public
boolean handledBy(AsyncEventHandler handler)
throws IllegalStateException
```

Parameters

handler to add to the Timer

Throws

IllegalStateException when this `Timer` has been *destroyed.*

Returns

`true` when `handler` is associated with `this`, otherwise `false`.

createReleaseParameters

Create a ReleaseParameters[92] object appropriate to the timing characteristics of this event. The default is the most pessimistic: AperiodicParameters[93]. This is typically called by code that is setting up a `handler` for this event that will fill in the parts of the release parameters for which it has values, e.g. cost.

Signature

```
public
javax.realtime.ReleaseParameters createReleaseParameters()
throws IllegalStateException
```

Throws

IllegalStateException when this `Timer` has been *destroyed*.

Returns

A newly created ReleaseParameters[94] object.

addHandler(AsyncEventHandler)

Add a handler to release upon fire.

Available since RTSJ 1.0.1

Signature

```
public
void addHandler(AsyncEventHandler handler)
throws IllegalStateException, IllegalAssignmentError
```

Parameters

handler to add to the Timer

Throws

IllegalStateException when this `Timer` has been *destroyed*.

IllegalAssignmentError @inheritDoc

setHandler(AsyncEventHandler)

Set a handler and remove all others.

[92]Section 6.4.2.8
[93]Section 6.4.2.2
[94]Section 6.4.2.8

Available since RTSJ 1.0.1

Signature

```
public
void setHandler(AsyncEventHandler handler)
throws IllegalStateException, IllegalAssignmentError
```

Parameters

handler to add to the Timer

Throws

IllegalStateException when this `Timer` has been *destroyed*.

IllegalAssignmentError @inheritDoc

removeHandler(AsyncEventHandler)

Remove the given handler.

Available since RTSJ 1.0.1

Signature

```
public
void removeHandler(AsyncEventHandler handler)
throws IllegalStateException
```

Parameters

handler to add to the Timer

Throws

IllegalStateException when this `Timer` has been *destroyed*.

enable

Re-enable this timer after it has been *disabled*. (See `Timer.disable()`[95].) When the `Timer` is already *enabled*, this method does nothing. When the `Timer` is *not-active*, this method does nothing.

Signature

```
public
```

[95] Section 10.4.2.5.4

```
void enable()
throws IllegalStateException
```

Throws

IllegalStateException when this `Timer` has been *destroyed*.

disable

Disable this timer, preventing it from firing. It may subsequently be re-*enabled*. When the timer is *disabled* when its fire time occurs then it will not release its handlers. However, a *disabled* timer created using an instance of `RelativeTime` for its time parameter continues to count while it is *disabled*, and no changes take place in a *disabled* timer created using an instance of `AbsoluteTime`, in both cases the potential firing is simply masked, or skipped. When the timer is subsequently re-*enabled* before its fire time and it is *enabled* when its fire time occurs, then it will fire. It is important to note that this method does not delay the time before a possible firing. For example, when the timer is set to fire at time 42 and the `disable()` is called at time 30 and `enable()` is called at time 40 the firing will occur at time 42 (not time 52). These semantics imply also that firings are not queued. Using the above example, when enable was called at time 43 no firing will occur, since at time 42 `this` was *disabled*. When the `Timer` is already *disabled*, whether it is *active* or *inactive*, this method does nothing.

Signature

```
public
void disable()
throws IllegalStateException
```

Throws

IllegalStateException when this `Timer` has been *destroyed*.

start

Start this timer. A timer starts measuring time from when it is started; this method makes the timer *active* and *enabled*.

Signature

```
public
void start()
throws IllegalStateException
```

Throws

IllegalStateException when this `Timer` has been *destroyed*, or when this timer is already *active.*

start(boolean)

Start this timer. A timer starts measuring time from when it is started. When `disabled` is `true` start the timer making it *active* in a *disabled* state. When `disabled` is `false` this method behaves like the `start()` method.
Available since RTSJ 1.0.1

Signature

```
public
void start(boolean disabled)
throws IllegalStateException
```

Parameters

disabled When `true`, the timer will be *active* but *disabled* after it is started. When `false` this method behaves like the `start()` method.

Throws

IllegalStateException when this `Timer` has been *destroyed*, or when this timer is *active.*

stop

Stops a timer when it is *active* and changes its state to *inactive* and *disabled.*

Signature

```
public
boolean stop()
throws IllegalStateException
```

Throws

IllegalStateException when this `Timer` has been *destroyed.*

Returns

`true` when `this` was *enabled* and `false` otherwise.

reschedule(javax.realtime.HighResolutionTime<?>)

Change the scheduled time for this event. This method can take either an `AbsoluteTime` or a `RelativeTime` for its argument, and the `Timer` will behave as if created using that type for its `time` parameter. The rescheduling will take place between the invocation and the return of the method.

NOTE: While the scheduled time is changed as described above, the rescheduling itself is applied only on the first firing (or on the first skipping when *disabled*) of a timer's activation. When `reschedule` is invoked after the current activation timer's firing, then the rescheduled `time` will be effective only upon the next `start` or `startDisabled` command (which may need to be preceded by a `stop` command).

When `reschedule` is invoked with a `RelativeTime time` on an *active* timer before its first firing/skipping, then the rescheduled firing/skipping `time` is relative to the time of invocation.

Signature

```
public
void reschedule(javax.realtime.HighResolutionTime<?> time)
throws IllegalStateException, IllegalArgumentException
```

Parameters

time The time to reschedule for this event firing. When `time` is `null`, the previous time is still the time used for the `Timer` firing. The clock associated with the parameter `time` is always ignored.

Throws

IllegalArgumentException when `time` is a negative `RelativeTime` value.
IllegalStateException when this `Timer` has been *destroyed*.

fire

Should not be called except for emulation. The fire method is reserved for the use of the timer. It releases all handlers when `this` is enabled and does nothing, otherwise.

Signature

```
public
void fire()
```

10.5 Rationale

Clocks differ because of monotonicity, synchronization, jitter, stability, accuracy, precision, and resolution. There are many possible subclasses of clocks: realtime clocks, user time clocks, simulation time clocks, wall clocks.

The idea of using multiple clocks may at first seem unusual, but it enables the accommodation of difference kinds of clocks and as a possible resource allocation strategy. Consider a realtime system where the natural events of the system have different tolerances for jitter. Assume the system functions properly if event A is triggered within plus or minus 100 seconds of the actual time it should occur but event B must be triggered within 100 microseconds of its actual time. Further assume, without loss of generality, that events A and B are periodic. An application could then create two instances of `PeriodicTimer` based on two clocks. The timer for event B should be based on a `Clock` which checks its queue at least every 100 microseconds but the timer for event A could be based on a `Clock` that checked its queue only every 100 seconds. This use of two clocks reduces the queue size of the accurate clock and thus queue management overhead is reduced.

The importance of the use of one-shot timers for time-out behavior and the vagaries in the execution of code prior to starting the timer for short time-outs dictate that the triggering of the timer should be guaranteed. The problem is exacerbated for periodic timers where the importance of the periodic triggering outweighs the precision of the start time. In such cases, it is also convenient to allow, for example, a relative time of zero to be used as the start time.

Clock resolution is a complicated topic, and clock implementations may have differing precissions for different purposes. For example, a clock for interacting with humans need much less precission than for controlling the openning and closing of values on an internal combustion engine. In this case, their relationship to wall clock time may vary as well.

The precision of time returned by a hardware clock device when queried may be greater than the precision at which that device can supply interrupts. (Consider, for example, a high precision off-chip realtime clock device connected via a shared serial bus.) A different device may provide pulse-per-second interrupts of very high precision, but be unable to interrupt on any other interval. The `RTSJ Clock` class provides two representation of precision: `getDrivePrecission()` and `getQueryPrecission` inherited from `Chronograph`. Clocks should behave as if their tick (`setAlarm()`) precision is the same as returned by `getResolution()`.

Chapter 11

Alternative Memory Areas

11.1 Overview

Conventional Java uses a single heap for storing all objects. The thread stacks hold only primative objects and references to objects. This is fine for desktop and server systems, where there are no realtime, locality, or isolation requirements. For most realtime systems, a single heap will a deterministic garbage collector is usually also sufficient. For other situations, this specification defines classes directly related to memory and memory management. These classes provide a more generalized means of memory management than available in a conventional Java VM.

In conventional Java, all of the memory needed for the allocation of an object is taken from a garbage-collected heap. The `RTSJ` generalizes the concept of a heap to that of a *memory area*. A memory area consists of two components: a Java object that manages the memory area and the *backing memory*, which is the actual region of memory from which objects are allocated. Every thread and schedulable has a *current allocation context*. This context is the memory area which is managing the backing memory that will be used when the thread/schedulable requests memory allocation using the Java "new" operator.

There are three types of memory area, distinguished by object lifetime semantics, defined by the `RTSJ`.

- Heap memory—the Java heap. Unreferenced objects are collected by a garbage collector. Individual schedulables can specify their rate of allocation of objects on the heap.
- Immortal memory—an area defined by the JVM in which allocated objects might never be collected. Access to the memory area must be independent of garbage collection activity. Individual schedulables can specify the maximum amount of memory they need in immortal memory.
- Scoped memory—multiple areas that can be created by the application; ob-

jects are collected in scoped memory when there are no schedulables currently active in that area. These allow objects with well-defined lifetimes to be created and efficiently collected in an easily identified group.

Given that objects can now be created in multiple memory areas, it is necessary to ensure that an object cannot reference another object that might be collected at an earlier time. For example, an object in immortal memory (that is never collected) must not be allowed to reference an object in scoped memory. This is because the scoped memory object will be collected when there is no schedulable active in its associated backing memory, rendering the immortal object's reference to the scoped memory object invalid. For this reason, the `RTSJ` defines some memory assignment rules that are checked by the JVM on every object assignment. If the program violates the assignment rules, an exception is thrown.

11.1.1 Physical Memory

In embedded systems it is often the case that multiple directly addressable memory types are available to the application. The JVM implementer may require the VM to be portable between systems within the same processor family. The VM, therefore, may have detailed knowledge of the underlying memory architecture. It is primarily concerned with the standard Random-Access Memory (RAM) provided to it by the host operating system. The `RTSJ`, therefore, provides a framework with which the embedded systems integrator can define memory characteristics and specify ranges of physical addresses that support those memory characteristics. Physical memory regions can be allocated as either immortal or scoped memory areas, as follows:

- Physical immortal memory—multiple immortal memory areas that can be created by the application such that their associated backing memory areas have specified physical and virtual memory characteristics. For example, the application could specify that the physical characteristics of the backing store should be Static RAM (SRAM) and that it should be mapped by the JVM into virtual memory that is never paged out to disk.
- Physical scoped memory—multiple scoped memory areas that can be created by the application such that their associated backing store has specified physical and virtual memory characteristics.

This physical memory model is based on two constraints:

- Java objects can only be allocated in a memory area if the physical backing memory supports the Java Memory Model (JMM) without the JVM having to perform any operation additional to those that it performs when accessing the main RAM for the host machine. No extra compiler or JVM interactions shall be required. Hence memory regions (such as EEPROM) that potentially require special hardware instructions to perform write operations cannot be used as the backing store for physical memory areas. Similarly, non-volatile memory cannot be used, as object lifetimes in such an area may be longer than the lifetime of the VM. Although memory having such characteristics incompatible with the JMM are prohibited from being used as backing stores for object allocation, they can contain objects of primitive Java types and be accessed via the `RTSJ` Raw Memory facilities (see Section 12.3.1).

- Any API must delegate detailed knowledge of the memory architecture to the programmer/integrator of the specific embedded system to be implemented. The model assumes that the programmer is aware of the memory map, either through some native operating system interface[1] or from some property file read at program initialization time.

The `RTSJ` defines a *physical memory manager*, which maintains a mapping between physical memory characteristics and the associated physical addresses of memory that support those characteristics. The physical memory manager has no knowledge of the meaning of the physical characteristics. It only provides a look-up service and keeps track of which physical memory has been allocated to a physical memory area's backing store by the application. The physical memory manager does, however, have detailed knowledge of the types of virtual memory it can support. It advertises this knowledge to the application. For example, it knows if the VM can lock memory pages into memory to ensure that they are never swapped out to disk. The application can then request that the physical memory manager create an association between physical memory with certain characteristics and a virtual memory type (for example, SRAM that is permanently resident in memory). The physical memory manager creates a *filter* to represent this association. These filters can then be used in the constructors to physical immortal and scoped memory areas to ensure that the backing memory has the required properties.

11.1.2 Stacked Memory

Systems that must both maintain predictable memory performance over a long period of time and allocate and release memory at runtime must be able to characterize and control both internal and external fragmentation. The `RTSJ` provides scoped memory for safe, application-driven allocation and release of memory, but the bare scoped memory interface (*e.g.,* `LTMemory`) leaves sufficient ambiguity in specification that using it to create and release scopes at runtime in a fragmentation-free manner may depend on the VM platform. The `StackedMemory` class provides a safe interface for creating and releasing scopes with a set of rules under which the VM must guarantee fragmentation-free behavior with predictable memory overhead. These

[1] For example, the *Advanced Configuration and Power Interface* (ACPI) specification is an open standard for device configuration and power management by the operating system. The ACPI defines platform-independent interfaces for hardware discovery, configuration, power management and monitoring. See http://www.acpi.info/

guarantees are provided by constraining the order in which an application may enter StackedMemory areas, as well as the manner in which they may be arranged on the scope stack. These constraints are to be enforced by the implementation.

A StackedMemory area represents both a backing memory providing ScopedMemory semantics and a *backing memory container* from which the backing memory is drawn. The backing memory container may be further subdivided into additional backing memories and backing memory containers. Such divisions behave as if new containers are allocated contiguously from the bottom of the container, while new backing memories are allocated contiguously from the top, with containers and backing memories meeting when the container is completely occupied.

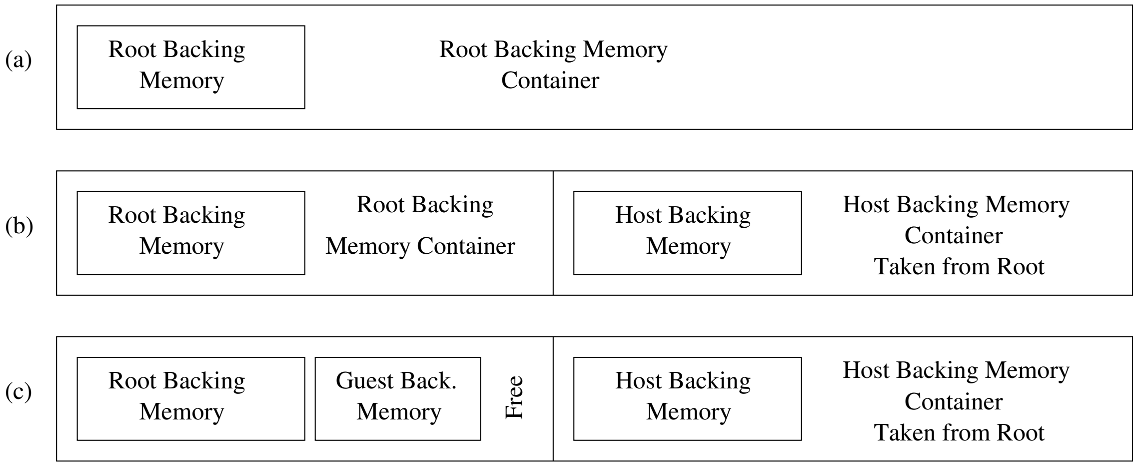
StackedMemory backing memory containers are explicitly created and sized, and have well-defined lifetimes similar to objects in a ScopedMemory area. A StackedMemory object can be created as either a *host*, which has its own backing memory container, or a *guest*, which draws its backing memory directly from its parent's backing memory container. When a StackedMemory object is created in an allocation context other than StackedMemory, it is necessarily a host and we call it a *root* StackedMemory. In this case, its backing memory container is drawn from a notional global backing memory container. Applications should assume that a root StackedMemory's object is immortal, and implementations need not release it even if it is finalized. When a StackedMemory object is created in another StackedMemory's allocation context, it may be created as either a host or guest, as illustrated in Figure 11.1. When it is created as a host, its backing memory container is drawn from its parent area's container, and its backing memory is created in the newly-divided container. When it is created as a guest, its backing memory is created in its parent's backing memory container.

Object lifetimes for objects allocated in StackedMemory backing memories are the same as those in ScopedMemory allocation areas. When a StackedMemory object itself is finalized, its backing memory is returned to the container from which it was drawn, and in the case of host StackedMemory areas, the associated backing memory container is also returned to the parent's container. As previously mentioned, root StackedMemory backing memory containers are effectively immortal. Additionally, the backing memory of a StackedMemory can be resized under certain conditions. These semantics allow the memory represented by a root StackedMemory backing memory container to be partitioned and re-partitioned as the application requires, without danger of fragmentation and without requiring memory allocation external to the container to track the partitioning.

In order to preserve the fragmentation-free nature of this contract, certain rules are enforced, and certain additional rules must be observed by the application. The rules that are enforced by the StackedMemory infrastructure are:

- A non-root StackedMemory area can only be entered from the same allocation

Figure 11.1: Manipulation of `StackedMemory` Areas



context in which it was created.

- A `StackedMemory` area may have at most one direct child in the scope stack that is a guest `StackedMemory` area.
- A `StackedMemory` object cannot be created from another `StackedMemory` allocation context unless it is allocated from that area's backing memory.
- A `StackedMemory`'s backing memory cannot be resized if there are non-finalized guest `StackedMemory` backing memories placed after it in the same backing memory container.

The additional rules that are not enforced by the infrastructure are not enforced because it may be desirable for an application to temporarily violate them (*e.g.*, when joining child threads in completion order, rather than creation order). They nevertheless must be observed at critical times in order to preserve fragmentation-free allocation. They are that, when a new `StackedMemory` is created as either a host or guest in a particular container, one of the following conditions must be met:

- The new `StackedMemory` area is a host, and there are existing non-finalized host `StackedMemory` descendants in the parent backing memory, but all such descendants were *created after* all non-finalized host descendants. (That is, effectively last-in first-out finalization.)
- The new `StackedMemory` area is a host, and there are no existing non-finalized host `StackedMemory` descendants in the parent backing memory.
- The new `StackedMemory` area is a guest.

Note that the first bullet point effectively can be violated only if the parent has *both* host and guest children.

Figure 11.1 graphically depicts the behavior of `StackedMemory` backing memory containers and backing memories for a root `StackedMemory` as well as one host and one guest child `StackedMemory` under that root. A code fragment that could create the stack topology in Figure 11.1 is as follows. Assume that this fragment executes in an allocation context other than a `StackedMemory`, and that zero overhead is required for memory area creation. (Implementations may require a constant amount of backing memory overhead for each StackedMemory area created in the store.)

```
// Create a StackedMemory with a 10 kB backing memory container and 2 kB backing memory
rootArea = new StackedMemory(2048, 10240); // (a)
rootArea.enter(new Runnable() {
    public void run() {
        // Create a host area with a 6 kB backing memory container and 2 kB backing memory
        hostArea = new StackedMemory(2048, 6144); // (b)
        // Create a guest area with a 2 kB backing memory
        guestArea = new StackedMemory(1536); // (c)
    }
});
```

Commented points (a), (b), and (c) correspond to their respective subfigures in Figure 11.1. At point (a), a root `StackedMemory` has been created with its 10 kB backing memory container drawn from the notional global container. It contains a 2 kB backing memory, which is then entered. With that backing memory as the current allocation context, a new host `StackedMemory` is created at (b), reserving 6 kB of the root `StackedMemory`'s backing memory container for its own use and creating a second 2 kB backing memory within that reservation. A new guest `StackedMemory` is then created at (c) in the root area (without entering the host child), occupying 1.5 kB of the remaining free 2 kB of the container in the root area. At this point, the root area's backing memory container is almost entirely occupied, with one 2 kB backing memory, one 1.5 kB backing memory, and a 6 kB host area backing memory container reservation, and 512 B of free backing memory container in between. The host `StackedMemory` created at (b) has 4 kB of its backing memory container remaining unoccupied in its reservation, which could be allocated to additional host or guest `StackedMemory` areas beneath it in the stack.

11.1.3 Summary

In summary, the classes and interfaces defined in this chapter enable

- the definition of regions of memory outside of the conventional Java heap;
- the definition of regions of scoped memory, that is, memory regions with a limited lifetime;
- the definition of regions of memory containing objects whose lifetime matches that of the application;
- the definition of regions of memory mapped to specific physical addresses with specific virtual memory characteristics;

- the specification of maximum memory area consumption and maximum allocation rates for individual schedulables;
- the programmer to query information characterizing the behavior of the garbage collection algorithm, and to some limited ability, alter the behavior of that algorithm.

11.2 Definitions

The following terms are used throughout this chapter.

Allocation Context—the place where a given task allocated new object. In conventional Java, this is the Java heap. This is just one type of `MemoryArea`, the base class representing all allocation contexts in the `RTSJ`.

Current allocation context—the memory area which will be used when object allocation is requested.

Backing store or backing memory—the area of memory that is managed by a `MemoryArea` and is logically separate from the Java heap.

Firable asynchronous event handler—An `AbstractAsyncEventHandler` is *firable* whenever there is an agent that can release it. This includes cases when the `AbstractAsyncEventHandler` is

- a miss handler, or overrun handler of a `RealtimeThread` instance that has been started but not yet terminated;
- a handler associated with an `AsyncEvent` that can be fired;
- a miss handler or overrun handler for an instance of `AbstractAsyncEventHandler`.

It excludes the final stage of scoped memory wrap-up.

Execution context—the memory area upon which execution is dependent. This includes the area a `Schedulable` or `ActiveEvent` is allocated. In order to prevent references from becoming invalid, the memory of an execution context may not be reclaimed. A memory area containing one of the following is such a context:

- `Thread` instances that have been started and have not terminated (including the `RealtimeThread` instances contained by `ActiveEventDispatcher` instances),
- `AbstractAsyncEvent` instances that have a least one associated `AbstractAsyncEventHandler`,
- `ActiveEvent` instances that are active,
- Firable `AbstractAsyncEventHandler` instance, and
- active threads of control which have the memory area enter on their stack.

Default initial memory area—The initial memory area for a schedulable is *default* when it is the memory area where the schedulable was created.

Portal—A reference inside a `ScopedMemory` instance settable on that instance which may reference an object in that area. A portal can be used to pass information between instances of `Schedulable` executing in that areas.

Scope stack—A stack of the memory areas the current `Schedulable` instance has entered in order of entry. in-use memory areas.

11.3 Semantics

The following list establishes the semantics that are applicable across the classes of this section. Semantics that apply to particular classes, constructors, methods, and fields will be found in the class description and the constructor, method, and field detail sections.

11.3.1 Allocation time

1. Some `MemoryArea` classes are required to have linear (in object size) allocation time. The linear time attribute requires that, ignoring performance variations due to hardware caches or similar optimizations and ignoring execution time of any static initializers, the execution time of `new` must be bounded by a polynomial, $f(n)$, where n is the size of the object and for all $n > 0$, $f(n) \leq Cn$ for some constant C.
2. Execution time of object constructors, and time spent in class loading and static initialization are not governed by bounds on object allocation in this specification, but setting default initial values for fields in the instance (as specified in *The Java Virtual Machine Specification*, Second Edition, section 2.5.1, "Each class variable, instance variable, and array component is initialized with a default value when it is created.") is considered part of object allocation and included in the time bound.

11.3.2 The allocation context

3. A memory area is represented by an instance of a subclass of the `MemoryArea` class. When a memory area, m, is entered by calling `m.enter` (or another method from the family of enter-like methods defined in `MemoryArea` or its subclasses), m becomes the *allocation context* of the current schedulable object. When control returns from the `enter` method, the allocation context is restored to the value it had immediately before `enter` was called.
4. When a memory area, m, is entered by calling m's `executeInArea` method, m becomes the current allocation context of the current schedulable. When

control returns from the `executeInArea` method, the allocation context is restored to the value it had before `executeInArea` was called.

5. The initial allocation context for a schedulable is the memory area that was designated the *initial memory area* when the schedulable was constructed. This initial allocation context becomes the current allocation context for that schedulable when the schedulable object first becomes eligible for execution. For instances of `AbstractAsyncEventHandler`, the initial allocation context is the same on each release; for realtime threads, in releases subsequent to the first, the allocation context is the same as it was when the realtime thread became *blocked-for-release-event*.
6. All object allocation through the `new` keyword will use the current allocation context, but note that allocation can be performed in a specific memory area using the `newInstance` and `newArray` methods.
7. Instances of `schedulables` behave as if they stored their memory area context in a structure called the *scope stack*. This structure is manipulated by the instantiation of a schedulables, and the following methods from `MemoryArea` and its subclasses: all the `enter` and `joinAndEnter` methods, `executeInArea`, and both `newInstance` methods. See the semantics in Maintaining the Scope Stack for details.
8. The scope stack is accessible through a set of static methods on `RealtimeThread`. These methods allow outer allocation contexts to be accessed by their index number. Memory areas on a scope stack may be referred to as *inner* or *outer* relative to other entries in that scope stack. An "outer scope" is further from the current allocation context on the current scope stack and has a lower index.
9. The `executeInArea, newInstance` and `newArray` methods, when invoked on an instance of `ScopedMemory` require that instance to be an outer allocation context on the current schedulable object's current scope stack.
10. An instance of `ScopedMemory` is said to be *in use* if it has a non-zero reference count as defined by semantic 17 below.

11.3.3 The Parent Scope

11. Instances of `ScopedMemory` have special semantics including definition of *parent*. If a `ScopedMemory` object is neither in use nor the initial memory area for a schedulable, it has no *parent* scope.
    - When a `ScopedMemory` object becomes in use, its parent is the nearest `ScopedMemory` object outside it on the current scope stack. If there is no outside `ScopedMemory` object in the current scope stack, the parent is the *primordial scope* which is not actually a memory area, but only a marker that constrains the parentage of `ScopedMemory` objects.

- At construction of a schedulable, if the initial memory area has no parent, the initial memory area is assigned the parent it will have when the schedulable is in execution. This rule determines the initial memory area's parent until the schedulable object is de-allocated or, in the case of a `RealtimeThread`, it completes execution.

12. Instances of `ScopedMemory` must satisfy the *single parent rule* which requires that each scoped memory has a unique parent as defined in semantic 11.

11.3.4 Memory areas and schedulables

13. Pushing a scoped memory onto a scope stack is always subject to the single parent rule.
14. Each schedulable has an initial memory area which is that object's initial allocation context. The default initial memory area is the current allocation context in effect during execution of the schedulable's constructor, but schedulables may supply constructors that override the default.
15. A Java thread cannot have a scope stack; consequently it can only be created and execute within heap or immortal memory. The thread starts execution with its allocation context set to the memory area containing the `Thread` object. An attempt to create a Java thread in a scoped memory area throws `IllegalAssignmentError`.
16. A Java thread may use `executeInArea`, and the `newInstance` and `newArray` methods from the `ImmortalMemory` and `HeapMemory` classes. These methods allow it to execute with an immortal current allocation context, but semantic 15 applies even during execution of these methods.

11.3.5 Scoped memory reference counting

17. Each instance of the class `ScopedMemory` or its subclasses must maintain a reference count which is greater than zero if and only if either:
    - the scoped memory area is the current allocation context or an outer allocation context for one or more *execution contexts*; or else
    - the scoped memory area contains the *fireable* `AsyncEventHandler` that was created with the `pinInitialmemoryArea` flag set to True.
18. Each instance of the `PinnableMemory` class must support a pinned count. This count is incremented for each call of the `pin` method and decremented for each call of the `unpin` method. The count is always greater than or equal to zero (that is, calling the `unpin` method has no effect if the count equals zero).
19. When the reference count for an instance of the class `ScopedMemory` is ready to be decremented from one to zero and the pinned count (if present) is equal to zero, all unfinalized objects within that area are considered ready for fi-

nalization. If after the finalizers for all such unfinalized objects in the scoped memory area run to completion, the reference count for the memory area is still ready to be decremented to zero and the pinned count is still equal to zero, any newly created unfinalized objects are considered ready for finalization and the process is repeated until no new objects are created or the scoped memory's reference count is no longer ready to be decremented from one to zero. When the scope contains no unfinalized objects and its reference count is ready to be decremented from one to zero and the pinned count is equal to zero, any async event in the scope is no longer treated as a source of fireability for async event handlers. If that action causes object creation in the scope the finalization process resumes from the beginning, if the reference count is no longer ready to be decremented to zero the finalization process terminates, otherwise, the reference count is decremented to zero and the memory scope is emptied of all objects. The process of scope finalization starts when the scope's reference count is about to go to zero with a zero pin count and continues until the scope is emptied or the process is terminated because the reference count is no longer about to go to zero. The `RTSJ` implementation must behave effectively as if during the finalization process the SO executing the finalization of a scope held a synchronized lock that must also be acquired to increase the reference count when entering the scope, to increase the reference count during startup for a thread with the finalizing scope as its non-default initial memory area, and to increase the reference count while making fireable an AEH with the scope as its non-default initial memory area. Although the steps in scope finalization are ordered no order is specified for finalization of objects or for disarming fireability of AEHs. The objects may be processed in any order or concurrently, but at no time may a scope's reference count be reduced to zero while it has one or more child scopes. (This semantic is a special case of the finalization implementation specified in *The Java Language Specification*, second edition, section 12.6.1.)

20. When the pinned count is ready to go to zero and the reference count is zero, all unfinalized objects within that area are considered ready for finalization, and the same semantics as 19 above applies.
21. Finalization may start when all unfinalized objects in the scope are ready for finalization. Finalizers are executed with the current allocation context set to the finalizing scope and are executed by the schedulable in control of the scope when its reference count is ready to be decremented from one to zero. If finalizers are executed because a realtime thread terminates or an `AsyncEventHandler` becomes non-fireable, that realtime thread or `AsyncEventHandler` is considered in control of the scope and must execute the finalizers.
22. From the time objects in a scope are deleted until the portal on the scope

is successfully set to a non-null value with `setPortal`, the value returned by `getPortal` on that scoped memory object must be `null`.

11.3.6 Immortal memory

23. Objects created in any immortal memory area are unexceptionally referenceable from all Java threads, and all schedulables, and the allocation and use of objects in immortal memory is never subject to garbage collection delays.
24. An implementation may execute finalizers for immortal objects when it determines that the application has terminated. Finalizers will be executed by a thread or schedulable whose current allocation context is not scoped memory. Regardless of any call to `runFinalizersOnExit`, except as required to support the base Java platform, the system need not execute finalizers for immortal objects that remain unfinalized when the JVM begins termination.
25. Class objects, the associated static memory, and interned Strings behave effectively as if they were allocated in immortal memory with respect to reference rules, assignment rules, and preemption delays by no-heap schedulables. Static initializers are executed effectively as if the current thread performed `ImmortalMemory.instance().executeInArea(r)` where `r` is a `Runnable` that executes the <`clinit`> method of the class being initialized.

11.3.7 Maintaining referential integrity

26. Assignment rules placed on reference assignments prevent the creation of dangling references, and thus maintain the referential integrity of the Java runtime. The restrictions are listed in the following table:

Stored in Area	**Reference to Object in Heap**	**Reference to Object in Immortal**	**Reference to Object in Scoped**	**null**
Heap	Permit	Permit	Forbid	Permit
Immortal	Permit	Permit	Forbid	Permit
Scoped	Permit	Permit	Permit from same or less deeply nested scope	Permit
Local Variable	Permit	Permit	Permit	Permit

For this table, `ImmortalMemory` and `ImmortalPhysicalMemory` are equivalent, and all sub-classes of `ScopedMemory` are equivalent.

27. An implementation must ensure that the above checks are performed for each assignment statement before the statement is executed, either by runtime checks or by static analysis of the application logic. Checks for operations on local

variables are not required because a potentially invalid reference would be captured by the other checks before it reached a local variable.

11.3.8 Object initialization

28. Static initializers run with the immortal memory area as their allocation context.
29. The current allocation context in a constructor for an object is the memory area in which the object is allocated. For `new`, this is the current allocation context when `new` was called. For members of the `m.newinstance` family, the current allocation context is memory area *m*.

11.3.9 Maintaining the Scope Stack

This section describes maintenance of a data structure that is called the *scope stack*. Implementations are not required to use a stack or implement the algorithms given here. It is only required that an implementation behave with respect to the ordering and accessibility of memory scopes effectively as if it implemented these algorithms. The scope stack is implicitly visible through the assignment rules, and the stack is explicitly visible through the static method `getOuterMemoryArea(int)` on `RealtimeThread`.

Four operations affect the scope stack: the `enter` methods defined in `MemoryArea` and its subclasses, instantiation of a new `Schedulable`, the `executeInArea` method in `MemoryArea`, and the new instance methods in `MemoryArea`.

- The memory area at the top of a schedulable object's scope stack is the schedulable's current allocation context.
- When a schedulable, t, creates a schedulable object, n_t, in a `ScopedMemory` object's allocation area, n_t acquires a copy of the scope stack associated with t at the time n_t is constructed including all entries from up to and including the memory area containing n_t. If n_t is created in heap, immortal, or immortal physical memory, n_t is created with a scope stack containing only heap, immortal, or immortal physical memory respectively. If n_t has a non-default initial memory area, *ima*, then *ima* is pushed on n_t's newly-created scope stack.
- When a memory area, `ma` is entered by calling a `ma.enter` method, `ma` is pushed on the scope stack and becomes the *allocation context* of the current schedulable object. When control returns from the `enter` method, the allocation context is popped from the scope stack
- When a memory area, `m`, is entered by calling `m`'s `executeInArea` method or one of the `m.newInstance` methods, the scope stack before the method call is preserved and replaced with a scope stack constructed as follows:

  – when `ma` is a scoped memory area, the new scope stack is a copy of the schedulable's previous scope stack up to and including `ma`, and
  – when `ma` is not a scoped memory area, the new scope stack includes only `ma`.

  When control returns from the `executeInArea` method, the scope stack is restored to the value it had before `ma.executeInArea` or `ma.newInstance` was called.

For the purposes of these algorithms, stacks grow *up*. One should also note that the representative algorithms ignore important issues like freeing objects in scopes.

- In every case, objects in a scoped memory area are eligible to be freed when the reference count for the area is zero after finalizers for that scope are run.
- Informally, any objects in a scoped memory area *must* be freed and their finalizers run before the reference count for the memory area is incremented from zero to one.

11.3.10 The `enter` method

For `ma.enter(logic)`:

```
push ma on the scope stack belonging to the current schedulable
   -- which may throw ScopedCycleException
execute logic.run method
pop ma from the scope stack
```

11.3.11 The `executeInArea` or `newInstance` methods

For `ma.executeInArea(logic)`, `ma.newInstance()`, or `ma.newArray()`:

```
when ma is an instance of heap immortal or ImmortalPhysicalMemory,
    start a new scope stack containing only ma.
    make the new scope stack the scope stack for the current
       schedulable.
else if ma is in the scope stack for the current schedulable,
    start a new scope stack containing ma and all
       scopes below ma on the scope stack.
    make the new scope stack the scope stack for the current
       schedulable.
else
    throw InaccessibleAreaException, execute logic.run,
        or construct the object.
    restore the previous scope stack for the current schedulable.
    discard the new scope stack.
end
```

11.3.12 Constructor methods for Schedulables

For construction of a schedulable in memory area `cma` with initial memory area of `ima`:

```
if cma is heap, immortal or ImmortalPhysicalMemory,
    create a new scope stack containing cma.
else
    start a new scope stack containing the entire
        current scope stack.

if ima != cma
    push ima on the new scope stack
        -- which may throw ScopedCycleException.
```

The above pseudocode illustrates a straightforward implementation of this specification's semantics, but any implementation that behaves effectively like this one with respect to reference count values of zero and one is permissible. An implementation may be eager or lazy in maintenance of its reference count provided that it correctly implements the semantics for reference counts of zero and one.

11.3.13 The Single Parent Rule

Every push of a scoped memory type on a scope stack requires reference to the single parent rule, this enforces the invariant that every scoped memory area has no more than one parent.

The parent of a scoped memory area is identified by the following rules:

- when the memory area is not currently on any scope stack, it has no parent;
- when the memory area is the first scoped memory area on a scope stack, i.e., was entered from `ImmortalMemory` or `Heap`, its parent is the *primordial scope*,
- otherwise, the parent is the first scoped memory area outside it on the scope stack, i.e., the scope from which this scope was entered.

Except for the primordial scope, which represents heap, immortal and immortal physical memory, only scoped memory areas are visible to the single parent rule.

The operational effect of the single parent rule is that when a scoped memory area has a parent, the only legal change to that value is to "no parent." Thus an ordering imposed by the first assignments of parents of a series of nested scoped memory areas is the only nesting order allowed until control leaves the scopes; then a new nesting order is possible. Thus, a schedulable attempting to enter a scope can only do so by entering in the established nesting order.

11.3.14 Scope Tree Maintenance

The single parent rule is enforced effectively as if there were a tree with the primordial scope (representing heap, immortal, and immortal physical memory) at its root, and other nodes corresponding to every scoped memory area that is currently on any schedulable's scope stack.

Each scoped memory has a reference to its parent memory area, `ma.parent`. The parent reference may indicate a specific scoped memory area, no parent, or the primordial parent.

If a scoped memory area is the non-default initial memory area of an async event handler, or the non-default initial memory area of a realtime thread that has not terminated, it is referred to as *pinned*.

11.3.14.1 On Scope Stack Push of ma

The following procedure could be used to maintain the scope tree and ensure that push operations on a schedulable's scope stack do not violate the single parent rule.

```
precondition: ma.parent is set to the correct parent (either a scoped
    memory area or the primordial scope) or to noParent.

  t.scopeStack is the scope stack of the current schedulable

  if ma is scoped,
      parent = findFirstScope(t.scopeStack)
  if ma.parent == noParent
      ma.parent = parent.
  else if ma.parent != parent
      throw ScopedCycleException.
  else
      t.scopeStack.push(ma).
```

`findFirstScope` is a convenience function that looks down the scope stack for the next entry that is a reference to an instance of `ScopedMemoryArea`.

```
findFirstScope(scopeStack)
{
  for s = top of scope stack to bottom of scope stack
  if s is an instance of scopedMemory
  return s return primordial scope
}
```

11.3.14.2 On Scope Stack Pop of ma

```
ma = t.scopeStack.pop.
if ma is scoped
    if !(ma.in_use || ma.pinned)
        ma.parent = noParent
```

11.4 Package javax.realtime

11.4.1 Interfaces

11.4.1.1 ChildScopeVisitor

This interface is used to visit the members of the set of scoped children of a memory area. An object implementing this interface is passed to the `MemoryArea.visitScopedChildren`[2] method.

11.4.1.1.1 Methods

visit(ScopedMemory)

Visit the members of the set of child scopes. It provides a means of accessing all live scopes contained in a memory area, even those to which no reference exits, such a `PinnableMemory`[3] that is pinned or another `ScopedMemory` that contains a `Schedulable`. The set may be concurrently modified by other tasks, but the view seen by the visitor may not be updated to reflect those changes. The following is guarantees even when the set is disturbed by other tasks:

- the visitor shall visit no member more than once,
- it shall visit only scopes that were a member of the set at some time during the enumeration of the set, and it shall visit all the scopes that are not deleted during the execution of the visitor.

Signature

```
public
java.lang.Object visit(ScopedMemory scope)
```

Parameters

scope The scoped memory area being visited.

Returns

Any object chosen by the application. When `visit` returns a non-null value, no more scopes are visited and the `MemoryArea.visitScopedChildren`[4] method

[2] Section 11.4.3.6.2
[3] Section 11.4.3.10
[4] Section 11.4.3.6.2

returns the value returned by `visit(ScopedMemory)`[5].

11.4.1.2 PhysicalMemoryCharacteristic

A tagging interface used to identify physical memory characteristics. Applications can give names to regions of memory that are described by `PhysicalMemoryModule`[6]. The names are defined by creating instances of this interface. For example, final static PhysicalMemoryCharacteristic STATIC_RAM = ...;
Available since RTSJ 2.0

11.4.1.3 PhysicalMemoryFilter

An interface to the physical memory filters. Filters are created by a factory in the `PhysicalMemoryManager`[7] class.
Available since RTSJ 2.0

11.4.1.3.1 Methods

---

setVMCharacteristics(javax.realtime.VirtualMemoryCharacteristic[])

Open issue: AJW I am not exactly sure whether this method is needed given we create the filter with the required VirtualMemoryCharacteristic Sets the virtual memory characteristics of this filter **End of open issue**

Signature

```
public
void setVMCharacteris-
tics(javax.realtime.VirtualMemoryCharacteristic[] required)
```

[5]Section 11.4.1.1.1
[6]Section 11.4.3.9
[7]Section 15.3.2.3

Parameters

required is an array of virtual memory characteristics required from the filter

11.4.1.4 VirtualMemoryCharacteristic

A tagging interface used to identify virtual memory characteristics. The PhysicalMemoryManager defines static public objects that implement this interface. Each instant represent a particular virtual memory characteristics supported by the hosting machine.

For example, public final static VirtualMemoryCharacteristic PERMANENTLY_RESIDENT

Available since RTSJ 2.0

11.4.2 Enumerations

11.4.2.1 NewPhysicalMemoryManager.CachingBehavior

Inheritance

java.lang.Object
 java.lang.Enum
  javax.realtime.NewPhysicalMemoryManager.CachingBehavior

11.4.2.1.1 Enumeration Constants

DISABLED

```
public static final DISABLED
```

WRITE_THROUGH

```
public static final WRITE_THROUGH
```

WRITE_BACK

```
public static final WRITE_BACK
```

11.4.2.1.2 Methods

---

values

Signature

```
public static
javax.realtime.NewPhysicalMemoryManager.CachingBehavior[]
values()
```

valueOf(String)

Signature

```
public static
javax.realtime.NewPhysicalMemoryManager.CachingBehavior
valueOf(String name)
```

11.4.2.2 NewPhysicalMemoryManager.PagingBehavior

Inheritance

java.lang.Object
  java.lang.Enum
    javax.realtime.NewPhysicalMemoryManager.PagingBehavior

11.4.2.2.1 Enumeration Constants

FIXED

```
public static final FIXED
```

SWAPPABLE

```
public static final SWAPPABLE
```

11.4.2.2.2 Methods

---

values

Signature

```
public static
javax.realtime.NewPhysicalMemoryManager.PagingBehavior[]
values()
```

valueOf(String)

Signature

```
public static
javax.realtime.NewPhysicalMemoryManager.PagingBehavior
valueOf(String name)
```

11.4.3 Classes

11.4.3.1 HeapMemory

The `HeapMemory` class is a singleton object that allows logic with a non-heap allocation context to allocate objects in the Java heap.

Inheritance
java.lang.Object

javax.realtime.MemoryArea
  javax.realtime.HeapMemory

11.4.3.1.1 Methods

---

enter

Associate this memory area with the current schedulable for the duration of the execution of the `run()` method of the instance of `Runnable` given in the constructor. During this period of execution, this memory area becomes the default allocation context until another default allocation context is selected (using `enter`, or `executeInArea`[8]) or the `enter` method exits.

Signature

`public`
`void enter()`

Throws

IllegalThreadStateException when the caller is a Java thread.
IllegalArgumentException @inheritDoc
MemoryAccessError when caller is a no-heap schedulable.

enter(Runnable)

Associate this memory area with the current schedulable for the duration of the execution of the `run()` method of the given `Runnable`. During this period of execution, this memory area becomes the default allocation context until another default allocation context is selected (using `enter`, or `executeInArea`[9]) or the `enter` method exits.

Signature

`public`
`void enter(Runnable logic)`

Parameters

logic The Runnable object whose `run()` method should be invoked.

---

[8] Section 11.4.3.1.1
[9] Section 11.4.3.1.1

Throws

MemoryAccessError when caller is a no-heap schedulable.
IllegalThreadStateException @inheritDoc
IllegalArgumentException @inheritDoc

instance

Returns a reference to the singleton instance of `HeapMemory`[10] representing the Java heap. The singleton instance of this class shall be allocated in the `ImmortalMemory`[11] area.

Signature

```
public static
javax.realtime.HeapMemory instance()
```

Returns

The singleton `HeapMemory`[12] object.

executeInArea(Runnable)

Execute the run method from the `logic` parameter using heap as the current allocation context. For a schedulable, this saves the current scope stack and replaces it with one consisting only of the `HeapMemory` instance; restoring the original scope stack upon completion.

Signature

```
public
void executeInArea(Runnable logic)
```

Parameters

logic The runnable object whose `run()` method should be executed.

Throws

IllegalArgumentException when `logic` is `null`.
MemoryAccessError when caller is a no-heap schedulable.

[10]Section 11.4.3.1
[11]Section 11.4.3.2
[12]Section 11.4.3.1

newArray(java.lang.Class<T>, int)

Allocate an array of the given type in this memory area. This method may be concurrently used by multiple threads.

Signature

```
public
T[] newArray(java.lang.Class<T> type, int number)
```

Parameters

type @inheritDoc
number @inheritDoc

Throws

MemoryAccessError when caller is a no-heap schedulable.
IllegalArgumentException @inheritDoc
OutOfMemoryError @inheritDoc

Returns

@inheritDoc

newInstance(java.lang.Class)

Allocate an object in this memory area. This method may be concurrently used by multiple threads.

Signature

```
public
java.lang.Object newInstance(java.lang.Class type)
throws IllegalAccessException, InstantiationException
```

Parameters

type @inheritDoc

Throws

MemoryAccessError when caller is a no-heap schedulable.
IllegalAccessException @inheritDoc
IllegalArgumentException @inheritDoc
ExceptionInInitializerError @inheritDoc
OutOfMemoryError @inheritDoc
InstantiationException @inheritDoc

Returns

@inheritDoc

newInstance(java.lang.reflect.Constructor, java.lang.Object[])

Allocate an object in this memory area. This method may be concurrently used by multiple threads.

Signature

```
public
java.lang.Object newInstance(java.lang.reflect.Constructor c,
java.lang.Object[] args)
throws IllegalAccessException, InstantiationException,
InvocationTargetException
```

Parameters

c T@inheritDoc

args @inheritDoc

Throws

MemoryAccessError when caller is a no-heap schedulable.

IllegalAccessException @inheritDoc

InstantiationException @inheritDoc

OutOfMemoryError @inheritDoc

IllegalArgumentException @inheritDoc

InvocationTargetException @inheritDoc

Returns

@inheritDoc

visitScopedChildren(ChildScopeVisitor)

Visit each scoped memory area who's parent is the primordial scope and was created in heap memory.

Signature

```
public
java.lang.Object visitScopedChildren(ChildScopeVisitor visitor)
```

Parameters

visitor invoke the `ChildScopeVisitor.visit(ScopedMemory)`[13] method for each member of the set of scoped memory areas that was created in heap memory and has the primordial scope as its parent.

[13] Section 11.4.1.1.1

Throws

IllegalArgumentException @inheritDoc

Returns

@inheritDoc

11.4.3.2 ImmortalMemory

`ImmortalMemory` is a memory resource that is unexceptionally available to all schedulables and Java threads for use and allocation.

An immortal object may not contain references to any form of scoped memory, e.g., `LTMemory`[14], `StackedMemory`[15], `PinnableMemory`[16], or `LTPhysicalMemory`[17].

Objects in immortal have the same states with respect to finalization as objects in the standard Java heap, but there is no assurance that immortal objects will be finalized even when the JVM is terminated.

Methods from `ImmortalMemory` should be overridden only by methods that use `super`.

Inheritance

java.lang.Object
  javax.realtime.MemoryArea
    javax.realtime.ImmortalMemory

11.4.3.2.1 Methods

instance

Returns a pointer to the singleton `ImmortalMemory`[18] object.

Signature

```
public static
javax.realtime.ImmortalMemory instance()
```

Returns

[14]Section 11.4.3.4
[15]Section 11.4.3.13
[16]Section 11.4.3.10
[17]Section 11.4.3.5
[18]Section 11.4.3.2

The singleton `ImmortalMemory`[19] object.

executeInArea(Runnable)

Execute the run method from the `logic` parameter using this memory area as the current allocation context. For a schedulable, this saves the current scope stack and replaces it with one consisting only of the `ImmortalMemory` instance; restoring the original scope stack upon completion.

Signature

```
public
void executeInArea(Runnable logic)
```

Parameters

logic The runnable object whose `run()` method should be executed.

Throws

IllegalArgumentException when `logic` is `null`.

visitScopedChildren(ChildScopeVisitor)

Visit each scoped memory area who's parent is the primordial scope and was created in this memory area.

Signature

```
public
java.lang.Object visitScopedChildren(ChildScopeVisitor visitor)
```

Parameters

visitor invoke the `ChildScopeVisitor.visit(ScopedMemory)`[20] method for each member of the set of scoped memory areas that was created in this immortal memory area and has the primordial scope as its parent.

Throws

IllegalArgumentException @inheritDoc

Returns

@inheritDoc

[19] Section 11.4.3.2
[20] Section 11.4.1.1.1

11.4.3.3 ImmortalPhysicalMemory

An instance of `ImmortalPhysicalMemory` allows objects to be allocated from a range of physical memory with particular attributes, determined by their memory type. This memory area has the same restrictive set of assignment rules as `ImmortalMemory`[21] memory areas, and may be used in any execution context where `ImmortalMemory` is appropriate.

No provision is made for sharing object in `ImmortalPhysicalMemory` with entities outside the JVM that creates them, and, while the memory backing an instance of `ImmortalPhysicalMemory` could be shared by multiple JVMs, the class does not support such sharing.

Methods from `ImmortalPhysicalMemory` should be overridden only by methods that use `super`.

Inheritance

java.lang.Object
  javax.realtime.MemoryArea
    javax.realtime.ImmortalPhysicalMemory

11.4.3.3.1 Constructors

ImmortalPhysicalMemory(PhysicalMemoryFilter, long)

Create a phyical immortal memory area of the specified type and size.
Available since RTSJ 2.0

Signature

```
public
  ImmortalPhysicalMemory(PhysicalMemoryFilter type, long size)
```

Parameters

[21]Section 11.4.3.2

type An instance of a physical memory filter that defines the the required characteristics of the physical memory and the required charcateristics of the virtual memory it is to mapped to.
size The size of memory required.

Throws

IllegalArgumentException when `size` is less than zero.
OutOfMemoryError when there is insufficient memory for the `ImmortalPhysicalMemory` object or for the backing memory.
SizeOutOfBoundsException when the `size` extends into an invalid range of memory.
SecurityException when the application doesn't have permissions to access physical memory or the given type of memory.

ImmortalPhysicalMemory(PhysicalMemoryFilter, SizeEstimator)

Create a phyical immortal memory area of the specified type and size.
Available since RTSJ 2.0

Signature

```
public
  ImmortalPhysicalMemory(PhysicalMemoryFilter type, SizeEstimator size)
```

Parameters

type An instance of a physical memory filter that defines the required characteristics of the physical memory and the required charcateristics of the virtual memory it is to mapped to.
size A size estimator for this memory area.

Throws

IllegalArgumentException when `size` is less than zero.
OutOfMemoryError when there is insufficient memory for the `ImmortalPhysicalMemory` object or for the backing memory.
SizeOutOfBoundsException when the `size` extends into an invalid range of memory.
SecurityException when the application doesn't have permissions to access physical memory or the given type of memory.

ImmortalPhysicalMemory(PhysicalMemoryFilter, long, Runnable)

Create a phyical immortal memory area of the specified type and size.
Available since RTSJ 2.0

Signature

```
public
  ImmortalPhysicalMemory(PhysicalMemoryFilter type, long size, Runnable logic)
```

Parameters

type An instance of a physical memory filter that defines the the required characteristics of the physical memory and the required charcateristics of the virtual memory it is to mapped to.
size The size of memory required.
logic The `run()` method of this object will be called whenever `MemoryArea.enter()`[22] is called. When `logic` is `null`, `logic` must be supplied when the memory area is entered.

Throws

IllegalArgumentException when `size` is less than zero.
OutOfMemoryError when there is insufficient memory for the `ImmortalPhysicalMemory` object or for the backing memory.
SizeOutOfBoundsException when the `size` extends into an invalid range of memory.
SecurityException when the application doesn't have permissions to access physical memory or the given type of memory.

ImmortalPhysicalMemory(PhysicalMemoryFilter, SizeEstimator, Runnable)

Create a phyical immortal memory area of the specified type and size.
Available since RTSJ 2.0

Signature

[22] Section 11.4.3.6.2

```
public
  ImmortalPhysicalMemory(PhysicalMemoryFilter type, SizeEstimator size, Runnable log
```

Parameters

type An instance of a physical memory filter that defines the the required characteristics of the physical memory and the required charcateristics of the virtual memory it is to mapped to.

size A size estimator for this memory area.

logic The `run()` method of this object will be called whenever `MemoryArea.enter()`[23] is called. When `logic` is `null`, `logic` must be supplied when the memory area is entered.

Throws

IllegalArgumentException when `size` is less than zero.

OutOfMemoryError when there is insufficient memory for the `ImmortalPhysicalMemory` object or for the backing memory.

SizeOutOfBoundsException when the `size` extends into an invalid range of memory.

SecurityException when the application doesn't have permissions to access physical memory or the given type of memory.

11.4.3.3.2 Methods

---

executeInArea(Runnable)

Open issue: AJW: Why is this method here? Execute the run method from the `logic` parameter using this memory area as the current allocation context. For a schedulable, this saves the current scope stack and replaces it with one consisting only of the `ImmortalMemory` instance; restoring the original scope stack upon completion. **End of open issue**

Signature

```
public
void executeInArea(Runnable logic)
```

Parameters

logic The runnable object whose `run()` method should be executed.

[23]Section 11.4.3.6.2

Throws

IllegalArgumentException when `logic` is `null`.

visitScopedChildren(ChildScopeVisitor)

Open issue: AJW: Why is this method here? Visit each scoped memory area who's parent is the primordial scope and was created in this immortal memory area. **End of open issue**

Signature

```
public
java.lang.Object visitScopedChildren(ChildScopeVisitor visitor)
```

Parameters

visitor invoke the `ChildScopeVisitor.visit(ScopedMemory)`[24] method for each member of the set of scoped memory areas that was created in this immortal memory area and has the primordial scope as its parent.

Throws

IllegalArgumentException @inheritDoc

Returns

@inheritDoc

11.4.3.4 LTMemory

`LTMemory` represents a memory area guaranteed by the system to have linear time allocation when memory consumption from the memory area is less than the memory area's *initial* size. Execution time for allocation is allowed to vary when memory consumption is between the initial size and the maximum size for the area. Furthermore, the underlying system is not required to guarantee that memory between initial and maximum will always be available.

The memory area described by a `LTMemory` instance does not exist in the Java heap, and is not subject to garbage collection. Thus, it is safe to use a `LTMemory` object as the initial memory area associated with a `NoHeapRealtimeThread`[25], or to enter the memory area using the `ScopedMemory.enter`[26] method within a `NoHeapRealtimeThread`[27].

[24]Section 11.4.1.1.1
[25]Section 15.3.2.1
[26]Section 11.4.3.11.2
[27]Section 15.3.2.1

Enough memory must be committed by the completion of the constructor to satisfy the initial memory requirement. (Committed means that this memory must always be available for allocation). The initial memory allocation must behave, with respect to successful allocation, as if it were contiguous; i.e., a correct implementation must guarantee that any sequence of object allocations that could ever succeed without exceeding a specified initial memory size will always succeed without exceeding that initial memory size and succeed for any instance of `LTMemory` with that initial memory size. (Note: to ensure that all requested memory is available set initial and maximum to the same value) Methods from `LTMemory` should be overridden only by methods that use `super`.

See Section MemoryArea

See Section ScopedMemory

See Section RealtimeThread

See Section NoHeapRealtimeThread

Inheritance

java.lang.Object
  javax.realtime.MemoryArea
    javax.realtime.ScopedMemory
      javax.realtime.LTMemory

11.4.3.4.1 Constructors

LTMemory(long, long)

Create an `LTMemory` of the given size.

Signature

```
public
  LTMemory(long initial, long maximum)
```

Parameters

initial The size in bytes of the memory to allocate for this area. This memory must be committed before the completion of the constructor.
maximum The size in bytes of the memory to allocate for this area.

Throws

IllegalArgumentException when `initial` is greater than `maximum`, or when `initial` or `maximum` is less than zero.
OutOfMemoryError when there is insufficient memory for the `LTMemory` object or for the backing memory.

LTMemory(long, long, Runnable)

Create an `LTMemory` of the given size.

Signature

```
public
  LTMemory(long initial, long maximum, Runnable logic)
```

Parameters

initial The size in bytes of the memory to allocate for this area. This memory must be committed before the completion of the constructor.
maximum The size in bytes of the memory to allocate for this area.
logic The `run()` of the given `Runnable` will be executed using `this` as its initial memory area. When `logic` is `null`, this constructor is equivalent to `LTMemory(long initial, long maximum)`[28].

Throws

IllegalArgumentException when `initial` is greater than `maximum`, or when `initial` or `maximum` is less than zero.
OutOfMemoryError when there is insufficient memory for the `LTMemory` object or for the backing memory.
IllegalAssignmentError when storing `logic` in `this` would violate the assignment rules.

LTMemory(SizeEstimator, SizeEstimator)

Create an `LTMemory` of the given size.

[28]Section 11.4.3.4.1

Signature

```
public
  LTMemory(SizeEstimator initial, SizeEstimator maximum)
```

Parameters

initial An instance of `SizeEstimator`[29] used to give an estimate of the initial size. This memory must be committed before the completion of the constructor.

maximum An instance of `SizeEstimator`[30] used to give an estimate for the maximum bytes to allocate for this area.

Throws

IllegalArgumentException when `initial` is `null`, `maximum` is `null`, `initial.getEstimate()` is greater than `maximum.getEstimate()`, or when `initial.getEstimate()` is less than zero.

OutOfMemoryError when there is insufficient memory for the `LTMemory` object or for the backing memory.

LTMemory(SizeEstimator, SizeEstimator, Runnable)

Create an `LTMemory` of the given size.

Signature

```
public
  LTMemory(SizeEstimator initial, SizeEstimator maximum, Runnable logic)
```

Parameters

initial An instance of `SizeEstimator`[31] used to give an estimate of the initial size. This memory must be committed before the completion of the constructor.

maximum An instance of `SizeEstimator`[32] used to give an estimate for the maximum bytes to allocate for this area.

[29] Section 11.4.3.12
[30] Section 11.4.3.12
[31] Section 11.4.3.12
[32] Section 11.4.3.12

logic The `run()` of the given `Runnable` will be executed using `this` as its initial memory area. When `logic` is `null`, this constructor is equivalent to `LTMemory(SizeEstimator initial, SizeEstimator maximum)`[33].

Throws

IllegalArgumentException when `initial` is `null`, `maximum` is `null`, `initial.getEstimate()` is greater than `maximum.getEstimate()`, or when `initial.getEstimate()` is less than zero.

OutOfMemoryError when there is insufficient memory for the `LTMemory` object or for the backing memory.

IllegalAssignmentError when storing `logic` in `this` would violate the assignment rules.

LTMemory(long)

Create an `LTMemory` of the given size. This constructor is equivalent to `LTMemory(size, size)`

Available since RTSJ 1.0.1

Signature

```
public
  LTMemory(long size)
```

Parameters

size The size in bytes of the memory to allocate for this area. This memory must be committed before the completion of the constructor.

Throws

IllegalArgumentException when `size` is less than zero.

OutOfMemoryError when there is insufficient memory for the `LTMemory` object or for the backing memory.

LTMemory(long, Runnable)

Create an `LTMemory` of the given size. This constructor is equivalent to `LTMemory(size, size, logic)`.

[33] Section 11.4.3.4.1

Available since RTSJ 1.0.1

Signature

```
public
  LTMemory(long size, Runnable logic)
```

Parameters

size The size in bytes of the memory to allocate for this area. This memory must be committed before the completion of the constructor.

logic The `run()` of the given `Runnable` will be executed using `this` as its initial memory area. When `logic` is `null`, this constructor is equivalent to `LTMemory(long size)`[34].

Throws

IllegalArgumentException when `size` is less than zero.

OutOfMemoryError when there is insufficient memory for the `LTMemory` object or for the backing memory.

IllegalAssignmentError when storing `logic` in `this` would violate the assignment rules.

LTMemory(SizeEstimator)

Create an `LTMemory` of the given size. This constructor is equivalent to `LTMemory(size, size)`.

Available since RTSJ 1.0.1

Signature

```
public
  LTMemory(SizeEstimator size)
```

Parameters

size An instance of `SizeEstimator`[35] used to give an estimate of the initial size. This memory must be committed before the completion of the constructor.

[34]Section 11.4.3.4.1
[35]Section 11.4.3.12

Throws

IllegalArgumentException when `size` is `null`, or `size.getEstimate()` is less than zero.
OutOfMemoryError when there is insufficient memory for the `LTMemory` object or for the backing memory.

LTMemory(SizeEstimator, Runnable)

Create an `LTMemory` of the given size.
Available since RTSJ 1.0.1

Signature

```
public
  LTMemory(SizeEstimator size, Runnable logic)
```

Parameters

size An instance of `SizeEstimator`[36] used to give an estimate of the initial size. This memory must be committed before the completion of the constructor.
logic The `run()` of the given `Runnable` will be executed using `this` as its initial memory area. When `logic` is `null`, this constructor is equivalent to `LTMemory(SizeEstimator initial)`[37].

Throws

IllegalArgumentException when `size` is `null`, or `size.getEstimate()` is less than zero.
OutOfMemoryError when there is insufficient memory for the `LTMemory` object or for the backing memory.
IllegalAssignmentError when storing `logic` in `this` would violate the assignment rules.

11.4.3.4.2 Methods

---

[36]Section 11.4.3.12
[37]Section 11.4.3.4.1

toString

Create a string representation of this object. The string is of the form
`(LTMemory) Scoped memory # num`
where `num` uniquely identifies the `LTMemory` area.

Signature

```
public
java.lang.String toString()
```

Returns

A string representing the value of `this`.

11.4.3.5 LTPhysicalMemory

An instance of `LTPhysicalMemory` allows objects to be allocated from a range of physical memory with particular attributes, determined by their memory type. This memory area has the same semantics as `ScopedMemory`[38] memory areas, and the same performance restrictions as `LTMemory`[39].

No provision is made for sharing object in `LTPhysicalMemory` with entities outside the JVM that creates them, and, while the memory backing an instance of `LTPhysicalMemory` could be shared by multiple JVMs, the class does not support such sharing.

Methods from `LTPhysicalMemory` should be overridden only by methods that use `super`.

Inheritance

java.lang.Object
  javax.realtime.MemoryArea
    javax.realtime.ScopedMemory
      javax.realtime.LTPhysicalMemory

11.4.3.5.1 Constructors

---

[38] Section 11.4.3.11
[39] Section 11.4.3.4

LTPhysicalMemory(PhysicalMemoryFilter, long)

Create a phyical immortal memory area of the specified type and size.
Available since RTSJ 2.0

Signature

```
public
  LTPhysicalMemory(PhysicalMemoryFilter type, long size)
```

Parameters

type An instance of a physical memory filter that defines the the required characteristics of the physical memory and the required charcateristics of the virtual memory it is to mapped to.
size The size of memory required.

Throws

IllegalArgumentException when `size` is less than zero.
OutOfMemoryError when there is insufficient memory for the `LTPhysicalMemory` object or for the backing memory.
SizeOutOfBoundsException when the `size` extends into an invalid range of memory.
SecurityException when the application doesn't have permissions to access physical memory or the given type of memory.

LTPhysicalMemory(PhysicalMemoryFilter, SizeEstimator)

Create a phyical immortal memory area of the specified type and size.
Available since RTSJ 2.0

Signature

```
public
  LTPhysicalMemory(PhysicalMemoryFilter type, SizeEstimator size)
```

Parameters

type An instance of a physical memory filter that defines the the required characteristics of the physical memory and the required charcateristics of the virtual memory it is to mapped to.
size A size estimator for this memory area.

Throws

IllegalArgumentException when `size` is less than zero.
OutOfMemoryError when there is insufficient memory for the `LTPhysicalMemory` object or for the backing memory.
SizeOutOfBoundsException when the `size` extends into an invalid range of memory.
SecurityException when the application doesn't have permissions to access physical memory or the given type of memory.

LTPhysicalMemory(PhysicalMemoryFilter, long, Runnable)

Create a phyical immortal memory area of the specified type and size.
Available since RTSJ 2.0

Signature

```
public
  LTPhysicalMemory(PhysicalMemoryFilter type, long size, Runnable logic)
```

Parameters

type An instance of a physical memory filter that defines the the required characteristics of the physical memory and the required charcateristics of the virtual memory it is to mapped to.
size The size of memory required.
logic The `run()` method of this object will be called whenever `MemoryArea.enter()`[40] is called. When `logic` is `null`, `logic` must be supplied when the memory area is entered.

Throws

IllegalArgumentException when `size` is less than zero.
OutOfMemoryError when there is insufficient memory for the `LTPhysicalMemory` object or for the backing memory.

[40] Section 11.4.3.6.2

SizeOutOfBoundsException when the `size` extends into an invalid range of memory.
SecurityException when the application doesn't have permissions to access physical memory or the given type of memory.

LTPhysicalMemory(PhysicalMemoryFilter, SizeEstimator, Runnable)

Create a phyical immortal memory area of the specified type and size.
Available since RTSJ 2.0

Signature

```
public
  LTPhysicalMemory(PhysicalMemoryFilter type, SizeEstimator size, Runnable logi
```

Parameters
type An instance of a physical memory filter that defines the the required characteristics of the physical memory and the required charcateristics of the virtual memory it is to mapped to.
size A size estimator for this memory area.
logic The `run()` method of this object will be called whenever `MemoryArea.enter()`[41] is called. When `logic` is `null`, `logic` must be supplied when the memory area is entered.
Throws
IllegalArgumentException when `size` is less than zero.
OutOfMemoryError when there is insufficient memory for the `LTPhysicalMemory` object or for the backing memory.
SizeOutOfBoundsException when the `size` extends into an invalid range of memory.
SecurityException when the application doesn't have permissions to access physical memory or the given type of memory.

11.4.3.5.2 Methods

---

[41]Section 11.4.3.6.2

toString

Creates a string describing this object. The string is of the form `(LTPhysicalMemory) Scoped memory # num` where `num` is a number that uniquely identifies this `LTPhysicalMemory` memory area. representing the value of `this`.

Signature

```
public
java.lang.String toString()
```

Returns

A string representing the value of `this`.

11.4.3.6 MemoryArea

`MemoryArea` is the abstract base class of all classes dealing with the representations of allocatable memory areas, including the immortal memory area, physical memory and scoped memory areas. This is an abstract class, but no method in this class is abstract. An application should not subclass `MemoryArea` without complete knowledge of its implementation details.

Inheritance

java.lang.Object
  javax.realtime.MemoryArea

11.4.3.6.1 Constructors

MemoryArea(long)

Create an instance of `MemoryArea`.

Signature

```
protected
  MemoryArea(long size)
```

Parameters

size The size of `MemoryArea` to allocate, in bytes.

Throws

IllegalArgumentException when `size` is less than zero.

OutOfMemoryError when there is insufficient memory for the `MemoryArea` object or for the backing memory.

MemoryArea(SizeEstimator)

Create an instance of `MemoryArea`.

Signature

```
protected
  MemoryArea(SizeEstimator size)
```

Parameters

size A `SizeEstimator`[42] object which indicates the amount of memory required by this `MemoryArea`.

Throws

IllegalArgumentException when the `size` parameter is `null`, or `size.getEstimate()` is negative.

OutOfMemoryError when there is insufficient memory for the `MemoryArea` object or for the backing memory.

MemoryArea(long, Runnable)

Create an instance of `MemoryArea`.

Signature

```
protected
  MemoryArea(long size, Runnable logic)
```

Parameters

size The size of `MemoryArea` to allocate, in bytes.

logic The `run()` method of this object will be called whenever `enter()`[43]

[42]Section 11.4.3.12

[43]Section 11.4.3.6.2

is called. When `logic` is `null`, this constructor is equivalent to `MemoryArea(long size)`.

Throws

IllegalArgumentException when the `size` parameter is less than zero.

OutOfMemoryError when there is insufficient memory for the `MemoryArea` object or for the backing memory.

IllegalAssignmentError when storing `logic` in `this` would violate the assignment rules.

MemoryArea(SizeEstimator, Runnable)

Create an instance of `MemoryArea`.

Signature

```
protected
  MemoryArea(SizeEstimator size, Runnable logic)
```

Parameters

size A `SizeEstimator` object which indicates the amount of memory required by this `MemoryArea`.

logic The `run()` method of this object will be called whenever `enter()`[44] is called. When `logic` is `null`, this constructor is equivalent to `MemoryArea(SizeEstimator size)`.

Throws

IllegalArgumentException when `size` is `null` or `size.getEstimate()` is negative.

OutOfMemoryError when there is insufficient memory for the `MemoryArea` object or for the backing memory.

IllegalAssignmentError when storing `logic` in `this` would violate the assignment rules.

11.4.3.6.2 Methods

[44] Section 11.4.3.6.2

enter

Associate this memory area with the current schedulable for the duration of the execution of the `run()` method of the instance of `Runnable` given in the constructor. During this period of execution, this memory area becomes the default allocation context until another default allocation context is selected (using `enter`, or `executeInArea`[45]) or the `enter` method exits.

Signature

```
public
void enter()
```

Throws

IllegalThreadStateException when the caller is a Java thread.

IllegalArgumentException when the caller is a schedulable and no non-null value for `logic` was supplied when the memory area was constructed.

ThrowBoundaryError Thrown when the JVM needs to propagate an exception allocated in `this` scope to (or through) the memory area of the caller. Storing a reference to that exception would cause an `IllegalAssignmentError`[46], so the JVM cannot be permitted to deliver the exception. The `ThrowBoundaryError`[47] is allocated in the current allocation context and contains information about the exception it replaces.

MemoryAccessError when caller is a no-heap schedulable and this memory area's logic value is allocated in heap memory.

enter(Runnable)

Associate this memory area with the current schedulable for the duration of the execution of the `run()` method of the given `Runnable`. During this period of execution, this memory area becomes the default allocation context until another default allocation context is selected (using `enter`, or `executeInArea`[48]) or the `enter` method exits.

Signature

```
public
```

[45] Section 11.4.3.6.2
[46] Section 14.2.3.3
[47] Section 14.2.3.6
[48] Section 11.4.3.6.2

`void enter(Runnable logic)`

Parameters

logic The Runnable object whose `run()` method should be invoked.

Throws

IllegalThreadStateException when the caller is a Java thread.

IllegalArgumentException when the caller is a schedulable and `logic` is `null`.

ThrowBoundaryError Thrown when the JVM needs to propagate an exception allocated in `this` scope to (or through) the memory area of the caller. Storing a reference to that exception would cause an `IllegalAssignmentError`[49], so the JVM cannot be permitted to deliver the exception. The `ThrowBoundaryError`[50] is allocated in the current allocation context and contains information about the exception it replaces.

enter(java.util.function.Supplier<T>)

Same as `enter(Runnable)`[51] except that the executed method is called `get` and an object is returned.

Signature

```
public
T enter(java.util.function.Supplier<T> logic)
```

Parameters

logic the object who's get method will be executed.

Returns

a result from the computation.

enter(BooleanSupplier)

Same as `enter(Runnable)`[52] except that the executed method is called `get` and a `boolean` is returned.

Signature

```
public
boolean enter(BooleanSupplier logic)
```

[49] Section 14.2.3.3
[50] Section 14.2.3.6
[51] Section 11.4.3.6.2
[52] Section 11.4.3.6.2

Parameters

logic the object who's get method will be executed.

Returns

a result from the computation.

enter(IntSupplier)

Same as `enter(Runnable)`[53] except that the executed method is called `get` and an `int` is returned.

Signature

```
public
int enter(IntSupplier logic)
```

Parameters

logic the object who's get method will be executed.

Returns

a result from the computation.

enter(LongSupplier)

Same as `enter(Runnable)`[54] except that the executed method is called `get` and a `long` is returned.

Signature

```
public
long enter(LongSupplier logic)
```

Parameters

logic the object who's get method will be executed.

Returns

a result from the computation.

enter(DoubleSupplier)

Same as `enter(Runnable)`[55] except that the executed method is called `get` and a `double` is returned.

[53] Section 11.4.3.6.2
[54] Section 11.4.3.6.2
[55] Section 11.4.3.6.2

Signature

```
public
double enter(DoubleSupplier logic)
```

Parameters

logic the object who's get method will be executed.

Returns

a result from the computation.

getMemoryArea(Object)

Gets the `MemoryArea` in which the given object is located.

Signature

```
public static
javax.realtime.MemoryArea getMemoryArea(Object object)
```

Throws

IllegalArgumentException when the value of `object` is `null`.

Returns

The instance of `MemoryArea` from which `object` was allocated.

memoryConsumed

For memory areas where memory is freed under program control this returns an exact count, in bytes, of the memory currently used by the system for the allocated objects. For memory areas (such as heap) where the definition of "used" is imprecise, this returns the best value it can generate in constant time.

Signature

```
public
long memoryConsumed()
```

Returns

The amount of memory consumed in bytes.

memoryRemaining

An approximation to the total amount of memory currently available for future allocated objects, measured in bytes.

Signature

```
public
long memoryRemaining()
```

Returns

The amount of remaining memory in bytes.

newArray(java.lang.Class<T>, int)

Allocate an array of the given type in this memory area. This method may be concurrently used by multiple threads.

Signature

```
public
T[] newArray(java.lang.Class<T> type, int number)
throws IllegalArgumentException, OutOfMemoryError
```

Parameters

type The class of the elements of the new array. To create an array of a primitive type use a `type` such as `Integer.TYPE` (which would call for an array of the primitive int type.)

number The number of elements in the new array.

Throws

IllegalArgumentException when `number` is less than zero, `type` is `null`, or `type` is `java.lang.Void.TYPE`.

OutOfMemoryError when space in the memory area is exhausted.

Returns

A new array of class type, of number elements.

newInstance(java.lang.Class<T>)

Allocate an object in this memory area. This method may be concurrently used by multiple threads.

Signature

```
public
```

```
T newInstance(java.lang.Class<T> type)
throws IllegalAccessException, IllegalArgumentException,
InstantiationException, OutOfMemoryError,
ExceptionInInitializerError, SecurityException
```

Parameters

type The class of which to create a new instance.

Throws

IllegalAccessException The class or initializer is inaccessible.

IllegalArgumentException when `type` is `null`.

InstantiationException when the specified class object could not be instantiated. Possible causes are: it is an interface, it is abstract, it is an array, or an exception was thrown by the constructor.

OutOfMemoryError when space in the memory area is exhausted.

ExceptionInInitializerError when an unexpected exception has occurred in a static initializer

SecurityException when the caller does not have permission to create a new instance.

Returns

A new instance of class `type`.

newInstance(java.lang.reflect.Constructor<T>, T[])

Allocate an object in this memory area. This method may be concurrently used by multiple threads.

Signature

```
public
T newInstance(java.lang.reflect.Constructor<T> c, T[] args)
throws ExceptionInInitializerError, IllegalAccessException,
IllegalArgumentException, InstantiationException,
InvocationTargetException, OutOfMemoryError
```

Parameters

c The constructor for the new instance.

args An array of arguments to pass to the constructor.

Throws

ExceptionInInitializerError when an unexpected exception has occurred in a static initializer

IllegalAccessException when the class or initializer is inaccessible under Java access control.

IllegalArgumentException when `c` is `null`, or the `args` array does not contain the number of arguments required by `c`. A `null` value of `args` is treated like an array of length 0.
InstantiationException when the specified class object could not be instantiated. Possible causes are: it is an interface, it is abstract, it is an array.
InvocationTargetException when the underlying constructor throws an exception.
OutOfMemoryError when space in the memory area is exhausted.

Returns

A new instance of the object constructed by `c`.

size

Query the size of the memory area. The returned value is the current size. Current size may be larger than initial size for those areas that are allowed to grow.

Signature

```
public
long size()
```

Returns

The size of the memory area in bytes.

executeInArea(Runnable)

Execute the run method from the `logic` parameter using this memory area as the current allocation context. The effect of `executeInArea` on the scope stack is specified in the subclasses of `MemoryArea`.

Signature

```
public
void executeInArea(Runnable logic)
throws IllegalArgumentException
```

Parameters

logic The runnable object whose `run()` method should be executed.

Throws

IllegalArgumentException when `logic` is `null`.

executeInArea(java.util.function.Supplier<T>)

Same as `executeInArea(Runnable)`[56] except that the executed method is called `get` and an object is returned.

Signature

```
public
T executeInArea(java.util.function.Supplier<T> logic)
```

Parameters

logic the object who's get method will be executed.

Returns

a result from the computation.

executeInArea(BooleanSupplier)

Same as `executeInArea(Runnable)`[57] except that the executed method is called `get` and a `boolean` is returned.

Signature

```
public
boolean executeInArea(BooleanSupplier logic)
```

Parameters

logic the object who's get method will be executed.

Returns

a result from the computation.

executeInArea(IntSupplier)

Same as `executeInArea(Runnable)`[58] except that the executed method is called `get` and an `int` is returned.

Signature

```
public
int executeInArea(IntSupplier logic)
```

[56] Section 11.4.3.6.2
[57] Section 11.4.3.6.2
[58] Section 11.4.3.6.2

Parameters

logic the object who's get method will be executed.

Returns

a result from the computation.

executeInArea(LongSupplier)

Same as `executeInArea(Runnable)`[59] except that the executed method is called `get` and a `long` is returned.

Signature

```
public
long executeInArea(LongSupplier logic)
```

Parameters

logic the object who's get method will be executed.

Returns

a result from the computation.

executeInArea(DoubleSupplier)

Same as `executeInArea(Runnable)`[60] except that the executed method is called `get` and a `double` is returned.

Signature

```
public
double executeInArea(DoubleSupplier logic)
```

Parameters

logic the object who's get method will be executed.

Returns

a result from the computation.

visitScopedChildren(ChildScopeVisitor)

Perform an action on all children scopes of this memory area, so long as the `ChildScopeVisitor.visit(LocalAllocationContext)` method returns `null`. When

[59] Section 11.4.3.6.2

[60] Section 11.4.3.6.2

that method returns an object, the visit is terminated and that object is returned by this method,

The set of children may be concurrently modified by other tasks, but the view seen by the visitor might not be updated to reflect those changes. The guarantees when the set is disturbed by other tasks are

- the visitor shall visit no member more than once,
- it shall visit only scopes that were a member of the set at some time during the enumeration of the set,
- it shall visit all the scopes that are not deleted during the enumeration of the set,
- it shall visit only scopes that were a member of the set at some time during the enumeration of the set, but need not visit all scopes that became a member of the set during the enumeration of the set, and
- it shall visit all the scopes that are not deleted during the execution of the visitor, but may also visit scopes that were deleted.

When execution of the visitor's visit method terminated abruptly by throwing an exception, then execution of visitScopedChildren also terminates abruptly by throwing the same exception.

Signature

```
public
java.lang.Object visitScopedChildren(ChildScopeVisitor visitor)
throws IllegalArgumentException
```

Parameters

visitor determines the action to be performed on each of the children scopes.

Throws

IllegalArgumentException when visitor is `null`.

Returns

`null` when all elements where visited and some object when the visit is forced to terminate at the end of visiting some element.

mayHoldReferenceTo

Determine whether an object `A` allocated in the memory area represented by `this` can hold a reference to an object `B` allocated in the current memory area.

Signature

```
public
boolean mayHoldReferenceTo()
```

Returns

`true` when `B` can be assigned to a field of `A`, otherwise `false`.

mayHoldReferenceTo(Object)

Determine whether an object `A` allocated in the memory area represented by `this` can hold a reference to the object `value`.

Signature

```
public
boolean mayHoldReferenceTo(Object value)
```

Parameters

value is the object to test.

Returns

`true` when `value` can be assigned to a field of `A`, otherwise `false`.

11.4.3.7 MemoryParameters

Memory parameters can be given on the constructor of `RealtimeThread`[61] and `AsyncEventHandler`[62]. These can be used both for the purposes of admission control by the scheduler and for the purposes of pacing the garbage collector (if any) to satisfy all of the schedulable memory allocation rates.

The limits in a `MemoryParameters` instance are enforced when a schedulable creates a new object, e.g., uses the `new` operation. When a schedulable exceeds its allocation or allocation rate limit, the error is handled as if the allocation failed because of insufficient memory. The object allocation throws an `OutOfMemoryError`.

When a reference to a `MemoryParameters` object is given as a parameter to a constructor, the `MemoryParameters` object becomes bound to the object being created. Changes to the values in the `MemoryParameters` object affect the constructed object. When given to more than one constructor, then changes to the values in the `MemoryParameters` object affect *all* of the associated objects. Note that this is a one-to-many relationship and *not* a many-to-many.

A `MemoryParameters` object may be shared, but that does not cause the memory budgets reflected by the parameter to be shared among the schedulables that are associated with the parameter object.

[61]Section 5.3.2.2

[62]Section 8.4.3.5

Caution: This class is explicitly unsafe for multithreading when being changed. Code that mutates instances of this class should synchronize at a higher level.

Inheritance

java.lang.Object
javax.realtime.MemoryParameters

Interfaces

`Cloneable`
`Serializable`

11.4.3.7.1 Fields

NO_MAX

`public static final NO_MAX`

Specifies no maximum limit.

11.4.3.7.2 Constructors

MemoryParameters(long, long)

Create a `MemoryParameters` object with the given values.

Signature

```
public
  MemoryParameters(long maxMemoryArea, long maxImmortal)
```

Parameters

maxMemoryArea A limit on the amount of memory the schedulable may allocate in its initial memory area. Units are in bytes. When zero, no allocation allowed in the memory area. To specify no limit, use `NO_MAX`.
maxImmortal A limit on the amount of memory the schedulable may allocate in the immortal area. Units are in bytes. When zero, no allocation allowed in immortal. To specify no limit, use `NO_MAX`.

Throws

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

MemoryParameters(long, long, long)

Create a `MemoryParameters` object with the given values.

Signature

```
public
  MemoryParameters(long maxMemoryArea, long maxImmortal, long allocationRate)
```

Parameters

maxMemoryArea A limit on the amount of memory the schedulable may allocate in its initial memory area. Units are in bytes. When zero, no allocation allowed in the memory area. To specify no limit, use `NO_MAX`.

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

allocationRate A limit on the rate of allocation in the heap. Units are in bytes per second of wall clock time. When `allocationRate` is zero, no allocation is allowed in the heap. To specify no limit, use `NO_MAX`. Measurement starts when the schedulable is first released for execution (not when it is constructed.) Enforcement of the allocation rate is an implementation option. When the implementation does not enforce allocation rate limits, it treats all non-zero allocation rate limits as `NO_MAX`.

Throws

IllegalArgumentException when any value other than positive. zero, or `NO_MAX` is passed as the value of `maxMemoryArea` or `maxImmortal`, or `allocationRate`.

11.4.3.7.3 Methods

---

clone

Return a clone of `this`. This method should behave effectively as if it constructed a new object with the visible values of `this`.

- The new object is in the current allocation context.
- `clone` does not copy any associations from `this` and it does not implicitly bind the new object to a SO.
-

Available since RTSJ 1.0.1

Signature

```
public
java.lang.Object clone()
```

getAllocationRate

Gets the limit on the rate of allocation in the heap. Units are in bytes per second.

Signature

```
public
long getAllocationRate()
```

Returns

The allocation rate in bytes per second. When zero, no allocation is allowed in the heap. When the returned value is `NO_MAX`[63] then the allocation rate on the heap is uncontrolled.

getMaxImmortal

Gets the limit on the amount of memory the schedulable may allocate in the immortal area. Units are in bytes.

Signature

```
public
long getMaxImmortal()
```

Returns

The limit on immortal memory allocation. When zero, no allocation is allowed in immortal memory. When the returned value is `NO_MAX`[64] then there is no limit for allocation in immortal memory.

[63] Section 11.4.3.7.1
[64] Section 11.4.3.7.1

getMaxMemoryArea

Gets the limit on the amount of memory the schedulable may allocate in its initial memory area. Units are in bytes.

Signature

```
public
long getMaxMemoryArea()
```

Returns

The allocation limit in the schedulable's initial memory area. When zero, no allocation is allowed in the initial memory area. When the returned value is `NO_MAX`[65] then there is no limit for allocation in the initial memory area.

setAllocationRate(long)

Sets the limit on the rate of allocation in the heap.

Signature

```
public
void setAllocationRate(long allocationRate)
```

Parameters

allocationRate Units are in bytes per second of wall-clock time. When `allocationRate` is zero, no allocation is allowed in the heap. To specify no limit, use `NO_MAX`. Measurement starts when the schedulable starts (not when it is constructed.) Enforcement of the allocation rate is an implementation option. When the implementation does not enforce allocation rate limits, it treats all non-zero allocation rate limits as `NO_MAX`.

Throws

IllegalArgumentException when any value other than positive, zero, or `NO_MAX` is passed as the value of `allocationRate`.

11.4.3.8 NewPhysicalMemoryManager

Each physical memory module can have more than one physical memory characteristic.

[65] Section 11.4.3.7.1

A physical memory characteristic can apply to many physical memory modules:

The `NewPhysicalMemoryManager` determines the physical addresses from the modules and keeps a relation between PhysicalMemoryModule <-> Physical Memory Addresses

The range of physical addresses of modules should not overlap.

To find a memory range that supports PMC A and PMC B uses set intersection modules(A) $
cap$ modules(B)

Need to consider exceptions that can be raised.

Available since RTSJ 2.0

Inheritance

java.lang.Object
javax.realtime.NewPhysicalMemoryManager

11.4.3.8.1 Constructors

NewPhysicalMemoryManager

Signature

```
public
  NewPhysicalMemoryManager()
```

11.4.3.8.2 Methods

associate(PhysicalMemoryCharacteristic, PhysicalMemoryModule)

Associates a programmer-defined name with a physical address range.

Signature

```
public static
void associate(PhysicalMemoryCharacteristic name,
PhysicalMemoryModule module)
```

Parameters

name is the physical memory characteristic. e.g STATIC_RAM.
module is the object representing a range of contiguous physical addresses

Throws

IllegalArgumentException when either name or module is null

associate(javax.realtime.PhysicalMemoryCharacteristic[], PhysicalMemoryModule)

Associates am array of programmer-defined names with a physical address range.

Signature

```
public static
void associate(javax.realtime.PhysicalMemoryCharacteristic[]
names, PhysicalMemoryModule module)
```

Parameters

names is the array of physical memory characteristics. e.g STATIC_RAM.
module is the object representing a range of contiguous physical addresses

Throws

IllegalArgumentException when either names or module is null

associate(PhysicalMemoryCharacteristic, javax.realtime.PhysicalMemory

Associates a programmer-defined name with an array of physical address ranges.

Signature

```
public static
void associate(PhysicalMemoryCharacteristic name,
javax.realtime.PhysicalMemoryModule[] modules)
```

Parameters

name is the physical memory characteristic. e.g STATIC_RAM.
modules is an array of objects each representing a range of contiguous physical addresses

Throws

IllegalArgumentException when either name or modules is null

createFilter(javax.realtime.PhysicalMemoryCharacteristic[], NewPhysicalMemoryManager.CachingBehavior, NewPhysicalMemoryManager.PagingBehavior)

Create a filter that will determine the appropriate place in physical memory that meets the required physical memory characteristics and map it into virtual memory with the given characteristic.

Open issue: name of exception **End of open issue**
Available since RTSJ 2.0

Signature

```
public static
javax.realtime.PhysicalMemoryFilter create-
Filter(javax.realtime.PhysicalMemoryCharacteristic[] PMcharac-
teristics, NewPhysicalMemoryManager.CachingBehavior cacheBehav-
ior, NewPhysicalMemoryManager.PagingBehavior pageBehavior)
```

Parameters

PMcharacteristics is an array of required physical memory characteristics.
cacheBehavior is the required caching behavior for mapped memory
pageBehavior is the required paging behavior for mapped memory

Throws

IllegalArgumentException when any of `PMcharacteristics` or `cacheBehavior` or `pageBehavior` is null
IllegalStateException when the required paging or caching behavior is not setable.

getSupportedCachingBehavior

Get the cacheing behaviors that are supported by this JVM
Available since RTSJ 2.0

Signature

```
public static
javax.realtime.NewPhysicalMemoryManager.CachingBehavior[]
getSupportedCachingBehavior()
```

Returns

an array of the supported cacheing behaviors.

getSupportedPagingBehavior

Get the paging behaviors that are supported by this JVM
Available since RTSJ 2.0

Signature

```
public static
javax.realtime.NewPhysicalMemoryManager.PagingBehavior[]
getSupportedPagingBehavior()
```

Returns

an array of the supported paging behaviors.

11.4.3.9 PhysicalMemoryModule

Enable an application to define a range of physical memory addresses.
Available since RTSJ 2.0

Inheritance

java.lang.Object
javax.realtime.PhysicalMemoryModule

11.4.3.9.1 Constructors

PhysicalMemoryModule(long, long)

Creates an instance representing a range of contiguous physical memory.

Signature

```
public
  PhysicalMemoryModule(long base, long length)
```

Parameters

base is a physical address

length is size of contiguous memory from that base

Throws

IllegalArgumentException when length is less than or equal to 0, or when base is less than 0 or when this module overlaps with another memory module.

SizeOutOfBounds when base + length is greater than the physical address range of the processor

11.4.3.9.2 Methods

getBase

Gets the base address of the contigous memory represented by this.

Signature

```
public
long getBase()
```

Returns

the base address

getLength

Gets the length of the contigous memory represented by this.

Signature

```
public
long getLength()
```

Returns

the length

11.4.3.10 PinnableMemory

Open issue: should this be a seperate class from `StackedMemory`[66]? **End of open issue**

Inheritance

java.lang.Object
  javax.realtime.MemoryArea
    javax.realtime.ScopedMemory
      javax.realtime.PinnableMemory

11.4.3.10.1 Constructors

PinnableMemory(long)

Signature

```
public
  PinnableMemory(long size)
```

PinnableMemory(SizeEstimator)

Signature

```
public
  PinnableMemory(SizeEstimator size)
```

11.4.3.10.2 Methods

[66]Section 11.4.3.13

pin

Signature

```
public
void pin()
```

unpin

Signature

```
public
void unpin()
```

isPinned

Signature

```
public
boolean isPinned()
```

getPinCount

Signature

```
public
int getPinCount()
```

joinPinned

Signature

```
public
void joinPinned()
throws InterruptedException
```

joinPinned(javax.realtime.HighResolutionTime<T>)

Signature

```
public
void joinPinned(javax.realtime.HighResolutionTime<T> timeIn)
throws InterruptedException
```

joinPinnedAndEnter(Runnable)

Signature

```
public
void joinPinnedAndEnter(Runnable logic)
throws InterruptedException, ScopedCycleException
```

joinPinnedAndEnter(Runnable, javax.realtime.HighResolutionTime<T>

Signature

```
public
void joinPinnedAndEnter(Runnable logic,
javax.realtime.HighResolutionTime<T> timeIn)
throws InterruptedException, ScopedCycleException
```

joinPinnedAndEnter

Signature

```
public
void joinPinnedAndEnter()
throws InterruptedException, IllegalThreadStateException,
ThrowBoundaryError, ScopedCycleException, MemoryAccessError
```

joinPinnedAndEnter(javax.realtime.HighResolutionTime<T>)

Signature

```
public
void joinPinnedAndEnter(javax.realtime.HighResolutionTime<T>
time)
throws InterruptedException, IllegalThreadStateException,
IllegalArgumentException, UnsupportedOperationException,
ThrowBoundaryError, ScopedCycleException, MemoryAccessError
```

11.4.3.11 ScopedMemory

`ScopedMemory` is the abstract base class of all classes dealing with representations of memory spaces which have a limited lifetime. In general, objects allocated in scoped memory are freed when (and only when) no schedulable object has access to the objects in the scoped memory.

A `ScopedMemory` area is a connection to a particular region of memory and reflects the current status of that memory. The object does not necessarily contain direct references to the region of memory. That is implementation dependent.

When a `ScopedMemory` area is instantiated, the object itself is allocated from the current memory allocation context, but the memory space that object represents (it's backing store) is allocated from memory that is not otherwise directly visible to Java code; e.g., it might be allocated with the C `malloc` function. This backing store behaves effectively as if it were allocated when the associated scoped memory object is constructed and freed at that scoped memory object's finalization.

The `ScopedMemory.enter`[67] method of `ScopedMemory` is one mechanism used to make a memory area the current allocation context. The other mechanism for activating a memory area is making it the initial memory area for a realtime thread or async event handler. Entry into the scope is accomplished, for example, by calling the method:

```
public void enter(Runnable logic)
```

where `logic` is a instance of `Runnable` whose `run()` method represents the entry point of the code that will run in the new scope. Exit from the scope occurs between the time the `runnable.run()` method completes and the time control returns from

[67]Section 11.4.3.11.2

the `enter` method. By default, allocations of objects within `runnable.run()` are taken from the backing store of the `ScopedMemory`.

`ScopedMemory` is an abstract class, but all specified methods include implementations. The responsibilities of `MemoryArea, ScopedMemory` and the classes that extend `ScopedMemory` are not specified. Application code should not extend `ScopedMemory` without detailed knowledge of its implementation.

Inheritance

java.lang.Object
  javax.realtime.MemoryArea
    javax.realtime.ScopedMemory

11.4.3.11.1 Constructors

ScopedMemory(long)

Create a new `ScopedMemory` area with the given parameters.

Signature

```
public
  ScopedMemory(long size)
```

Parameters

size of the new `ScopedMemory` area in bytes.

Throws

IllegalArgumentException when `size` is less than zero.
OutOfMemoryError when there is insufficient memory for the `ScopedMemory` object or for the backing memory.

ScopedMemory(long, Runnable)

Create a new `ScopedMemory` area with the given parameters.

Signature

```
public
  ScopedMemory(long size, Runnable logic)
```

Parameters

size The size of the new `ScopedMemory` area in bytes.

logic The `Runnable` to execute when this `ScopedMemory` is entered. When `logic` is `null`, this constructor is equivalent to constructing the memory area without a logic value.

Throws

IllegalArgumentException when `size` is less than zero.

IllegalAssignmentError when storing `logic` in `this` would violate the assignment rules.

OutOfMemoryError when there is insufficient memory for the `ScopedMemory` object or for the backing memory.

ScopedMemory(SizeEstimator)

Create a new `ScopedMemory` area with the given parameters.

Signature

```
public
  ScopedMemory(SizeEstimator size)
```

Parameters

size The size of the new `ScopedMemory` area estimated by an instance of `SizeEstimator`[68].

Throws

IllegalArgumentException when `size` is `null`, or `size.getEstimate()` is negative.

OutOfMemoryError when there is insufficient memory for the `ScopedMemory` object or for the backing memory.

ScopedMemory(SizeEstimator, Runnable)

Create a new `ScopedMemory` area with the given parameters.

[68] Section 11.4.3.12

Signature

```
public
  ScopedMemory(SizeEstimator size, Runnable logic)
```

Parameters

size The size of the new `ScopedMemory` area estimated by an instance of `SizeEstimator`[69].

logic The logic which will use the memory represented by `this` as its initial memory area. When `logic` is `null`, this constructor is equivalent to constructing the memory area without a logic value.

Throws

IllegalArgumentException when `size` is `null`, or `size.getEstimate()` is negative.

OutOfMemoryError when there is insufficient memory for the `ScopedMemory` object or for the backing memory.

IllegalAssignmentError when storing `logic` in `this` would violate the assignment rules.

11.4.3.11.2 Methods

---

enter

Associate this memory area with the current schedulable for the duration of the execution of the `run()` method of the instance of `Runnable` given in the constructor. During this period of execution, this memory area becomes the default allocation context until another default allocation context is selected (using `enter`, or `executeInArea`[70]) or the `enter` method exits.

Signature

```
public
void enter()
```

Throws

[69]Section 11.4.3.12
[70]Section 11.4.3.11.2

ScopedCycleException when this invocation would break the single parent rule.
ThrowBoundaryError Thrown when the JVM needs to propagate an exception allocated in `this` scope to (or through) the memory area of the caller. Storing a reference to that exception would cause an `IllegalAssignmentError`[71], so the JVM cannot be permitted to deliver the exception. The `ThrowBoundaryError`[72] is allocated in the current allocation context and contains information about the exception it replaces.
IllegalThreadStateException when the caller is a Java thread, or when this method is invoked during finalization of objects in scoped memory and entering this scoped memory area would force deletion of the SO that triggered finalization. This would include the scope containing the SO, and the scope (if any) containing the scope containing the SO.
IllegalArgumentException @inheritDoc
MemoryAccessError @inheritDoc

enter(Runnable)

Associate this memory area with the current schedulable for the duration of the execution of the `run()` method of the given `Runnable`. During this period of execution, this memory area becomes the default allocation context until another default allocation context is selected (using `enter`, or `executeInArea`[73]) or the `enter` method exits.

Signature

```
public
void enter(Runnable logic)
```

Parameters

logic @inheritDoc

Throws

ScopedCycleException when this invocation would break the single parent rule.
ThrowBoundaryError Thrown when the JVM needs to propagate an exception allocated in `this` scope to (or through) the memory area of the caller. Storing a reference to that exception would cause an `IllegalAssignmentError`[74], so the JVM cannot be permitted to deliver the exception. The `ThrowBoundaryError`[75] is allocated in the current allocation context and contains information

[71] Section 14.2.3.3
[72] Section 14.2.3.6
[73] Section 11.4.3.11.2
[74] Section 14.2.3.3
[75] Section 14.2.3.6

about the exception it replaces.
IllegalThreadStateException when the caller is a Java thread, or when this method is invoked during finalization of objects in scoped memory and entering this scoped memory area would force deletion of the SO that triggered finalization. This would include the scope containing the SO, and the scope (if any) containing the scope containing the SO.
IllegalArgumentException @inheritDoc

enter(java.util.function.Supplier<T>)

Same as `enter(Runnable)`[76] except that the executed method is called `get` and an object is returned.

Signature

```
public
T enter(java.util.function.Supplier<T> logic)
```

Parameters

logic the object who's get method will be executed.

Returns

a result from the computation.

enter(BooleanSupplier)

Same as `enter(Runnable)`[77] except that the executed method is called `get` and a `boolean` is returned.

Signature

```
public
boolean enter(BooleanSupplier logic)
```

Parameters

logic the object who's get method will be executed.

Returns

a result from the computation.

[76] Section 11.4.3.11.2
[77] Section 11.4.3.11.2

enter(IntSupplier)

Same as `enter(Runnable)`[78] except that the executed method is called `get` and an `int` is returned.

Signature

```
public
int enter(IntSupplier logic)
```

Parameters

logic the object who's get method will be executed.

Returns

a result from the computation.

enter(LongSupplier)

Same as `enter(Runnable)`[79] except that the executed method is called `get` and a `long` is returned.

Signature

```
public
long enter(LongSupplier logic)
```

Parameters

logic the object who's get method will be executed.

Returns

a result from the computation.

enter(DoubleSupplier)

Same as `enter(Runnable)`[80] except that the executed method is called `get` and a `double` is returned.

Signature

```
public
double enter(DoubleSupplier logic)
```

[78] Section 11.4.3.11.2

[79] Section 11.4.3.11.2

[80] Section 11.4.3.11.2

Parameters

logic the object who's get method will be executed.

Returns

a result from the computation.

executeInArea(Runnable)

Execute the run method from the `logic` parameter using this memory area as the current allocation context. This method behaves as if it moves the allocation context down the scope stack to the occurrence of `this`.

Signature

```
public
void executeInArea(Runnable logic)
```

Parameters

logic The runnable object whose `run()` method should be executed.

Throws

IllegalThreadStateException when the caller is a Java thread.

InaccessibleAreaException when the memory area is not in the schedulable's scope stack.

IllegalArgumentException when the caller is a schedulable and `logic` is `null`.

executeInArea(java.util.function.Supplier<T>)

Same as `executeInArea(Runnable)`[81] except that the executed method is called `get` and an object is returned.

Signature

```
public
T executeInArea(java.util.function.Supplier<T> logic)
```

Parameters

logic the object who's get method will be executed.

Returns

a result from the computation.

[81] Section 11.4.3.11.2

executeInArea(BooleanSupplier)

Same as `executeInArea(Runnable)`[82] except that the executed method is called `get` and a `boolean` is returned.

Signature

```
public
boolean executeInArea(BooleanSupplier logic)
```

Parameters

logic the object who's get method will be executed.

Returns

a result from the computation.

executeInArea(IntSupplier)

Same as `executeInArea(Runnable)`[83] except that the executed method is called `get` and an `int` is returned.

Signature

```
public
int executeInArea(IntSupplier logic)
```

Parameters

logic the object who's get method will be executed.

Returns

a result from the computation.

executeInArea(LongSupplier)

Same as `executeInArea(Runnable)`[84] except that the executed method is called `get` and a `long` is returned.

Signature

```
public
long executeInArea(LongSupplier logic)
```

[82] Section 11.4.3.11.2
[83] Section 11.4.3.11.2
[84] Section 11.4.3.11.2

Parameters

logic the object who's get method will be executed.

Returns

a result from the computation.

executeInArea(DoubleSupplier)

Same as `executeInArea(Runnable)`[85] except that the executed method is called `get` and a `double` is returned.

Signature

```
public
double executeInArea(DoubleSupplier logic)
```

Parameters

logic the object who's get method will be executed.

Returns

a result from the computation.

getPortal

Return a reference to the portal object in this instance of `ScopedMemory`.

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.

Signature

```
public
java.lang.Object getPortal()
```

Throws

IllegalAssignmentError when a reference to the portal object cannot be stored in the caller's allocation context; that is, when `this` is "inner" relative to the current allocation context or not on the caller's scope stack.

IllegalThreadStateException when the caller is a Java thread.

Returns

A reference to the portal object or `null` when there is no portal object. The portal value is always set to `null` when the contents of the memory are deleted.

[85]Section 11.4.3.11.2

getReferenceCount

Returns the reference count of this `ScopedMemory`.

Note: A reference count of 0 reliably means that the scope is not referenced, but other reference counts are subject to artifacts of lazy/eager maintenance by the implementation.

Signature

```
public
int getReferenceCount()
```

Returns

The reference count of this `ScopedMemory`.

join

Wait until the reference count of this `ScopedMemory` goes down to zero. Return immediately when the memory is unreferenced.

Signature

```
public
void join()
throws InterruptedException
```

Throws

InterruptedException When this schedulable is interrupted by `RealtimeThread.interrupt()`[86] or `AsynchronouslyInterruptedException.fire()`[87] while waiting for the reference count to go to zero.

IllegalThreadStateException when the caller is a Java thread.

join(javax.realtime.HighResolutionTime<T>)

Wait at most until the time designated by the `time` parameter for the reference count of this `ScopedMemory` to drop to zero. Return immediately when the memory area is unreferenced.

Since the time is expressed as a `HighResolutionTime`[88], this method is an accurate timer with nanosecond granularity. The actual resolution of the timer and

[86] Section 5.3.2.2.2
[87] Section 8.4.2.1.2
[88] Section 9.4.1.2

even the quantity it measures depends on the clock associated with `time`. The delay time may be relative or absolute. When relative, then the delay is the amount of time given by `time`, and measured by its associated clock. When absolute, then the delay is until the indicated value is reached by the clock. When the given absolute time is less than or equal to the current value of the clock, the call to `join` returns immediately.

Signature

```
public
void join(javax.realtime.HighResolutionTime<T> time)
throws InterruptedException
```

Parameters

time When this time is an absolute time, the wait is bounded by that point in time. When the time is a relative time (or a member of the `RationalTime` subclass of `RelativeTime`) the wait is bounded by a the specified interval from some time between the time `join` is called and the time it starts waiting for the reference count to reach zero.

Throws

InterruptedException When this schedulable is interrupted by `RealtimeThread.interrupt()`[89] or `AsynchronouslyInterruptedException.fire()`[90] while waiting for the reference count to go to zero.

IllegalThreadStateException when the caller is a Java thread.

IllegalArgumentException when the caller is a schedulable and `time` is `null`.

UnsupportedOperationException when the wait operation is not supported using the clock associated with `time`.

joinAndEnter

In the error-free case, `joinAndEnter` combines `join();enter();` such that no `enter()` from another schedulable can intervene between the two method invocations. The resulting method will wait for the reference count on this `ScopedMemory` to reach zero, then enter the `ScopedMemory` and execute the `run` method from `logic` passed in the constructor. When no instance of `Runnable` was passed to the memory area's constructor, the method throws `IllegalArgumentException` immediately.

When multiple threads are waiting in `joinAndEnter` family methods for a memory area, at most *one* of them will be released each time the reference count goes to zero.

[89] Section 5.3.2.2.2
[90] Section 8.4.2.1.2

Note that although `joinAndEnter` guarantees that the reference count is zero when the schedulable is released for entry, it does not guarantee that the reference count will remain one for any length of time. A subsequent `enter` could raise the reference count to two.

Signature

```
public
void joinAndEnter()
throws InterruptedException
```

Throws

InterruptedException When this schedulable is interrupted by `RealtimeThread.interrupt()`[91] or `AsynchronouslyInterruptedException.fire()`[92] while waiting for the reference count to go to zero.

IllegalThreadStateException when the caller is a Java thread, or when this method is invoked during finalization of objects in scoped memory and entering this scoped memory area would force deletion of the SO that triggered finalization. This would include the scope containing the SO, and the scope (if any) containing the scope containing the SO.

ThrowBoundaryError Thrown when the JVM needs to propagate an exception allocated in `this` scope to (or through) the memory area of the caller. Storing a reference to that exception would cause an `IllegalAssignmentError`[93], so the JVM cannot be permitted to deliver the exception. The `ThrowBoundaryError`[94] is allocated in the current allocation context and contains information about the exception it replaces.

ScopedCycleException when this invocation would break the single parent rule.

IllegalArgumentException when the caller is a schedulable and no non-null `logic` value was supplied to the memory area's constructor.

MemoryAccessError when caller is a non-heap schedulable and this memory area's logic value is allocated in heap memory.

joinAndEnter(javax.realtime.HighResolutionTime<T>)

In the error-free case, `joinAndEnter` combines `join();enter();` such that no `enter()` from another schedulable can intervene between the two method invocations. The resulting method will wait for the reference count on this `ScopedMemory`

[91] Section 5.3.2.2.2
[92] Section 8.4.2.1.2
[93] Section 14.2.3.3
[94] Section 14.2.3.6

to reach zero, or for the current time to reach the designated time, then enter the `ScopedMemory` and execute the `run` method from `Runnable` object passed to the constructor. When no instance of `Runnable` was passed to the memory area's constructor, the method throws `IllegalArgumentException` immediately. *

When multiple threads are waiting in `joinAndEnter` family methods for a memory area, at most *one* of them will be released each time the reference count goes to zero.

Since the time is expressed as a `HighResolutionTime`[95], this method has an accurate timer with nanosecond granularity. The actual resolution of the timer and even the quantity it measures depends on the clock associated with `time`. The delay time may be relative or absolute. When relative, then the calling thread is blocked for at most the amount of time given by `time`, and measured by its associated clock. When absolute, then the time delay is until the indicated value is reached by the clock. When the given absolute time is less than or equal to the current value of the clock, the call to `joinAndEnter` behaves effectively like `enter`[96].

Note that expiration of `time` may cause control to enter the memory area before its reference count has gone to zero.

Signature

```
public
void joinAndEnter(javax.realtime.HighResolutionTime<T> time)
throws InterruptedException
```

Parameters

time The time that bounds the wait.

Throws

ThrowBoundaryError Thrown when the JVM needs to propagate an exception allocated in `this` scope to (or through) the memory area of the caller. Storing a reference to that exception would cause an `IllegalAssignmentError`[97], so the JVM cannot be permitted to deliver the exception. The `ThrowBoundaryError`[98] is allocated in the current allocation context and contains information about the exception it replaces.

InterruptedException When this schedulable is interrupted by `RealtimeThread.interrupt()`[99] or `AsynchronouslyInterruptedException.fire()`[100] while waiting for the reference count to go to zero.

[95]Section 9.4.1.2
[96]Section 11.4.3.11.2
[97]Section 14.2.3.3
[98]Section 14.2.3.6
[99]Section 5.3.2.2.2
[100]Section 8.4.2.1.2

IllegalThreadStateException when the caller is a Java thread, or when this method is invoked during finalization of objects in scoped memory and entering this scoped memory area would force deletion of the SO that triggered finalization. This would include the scope containing the SO, and the scope (if any) containing the scope containing the SO.
ScopedCycleException when the caller is a schedulable and this invocation would break the single parent rule.
IllegalArgumentException when the caller is a schedulable, and `time` is `null` or no non-null `logic` value was supplied to the memory area's constructor.
UnsupportedOperationException when the wait operation is not supported using the clock associated with `time`.
MemoryAccessError when caller is a no-heap schedulable and this memory area's logic value is allocated in heap memory.

joinAndEnter(Runnable)

In the error-free case, `joinAndEnter` combines `join();enter();` such that no `enter()` from another schedulable can intervene between the two method invocations. The resulting method will wait for the reference count on this `ScopedMemory` to reach zero, then enter the `ScopedMemory` and execute the `run` method from `logic`

When `logic` is `null`, throw `IllegalArgumentException` immediately.

When multiple threads are waiting in `joinAndEnter` family methods for a memory area, at most *one* of them will be released each time the reference count goes to zero.

Note that although `joinAndEnter` guarantees that the reference count is zero when the schedulable is released for entry, it does not guarantee that the reference count will remain one for any length of time. A subsequent `enter` could raise the reference count to two.

Signature

```
public
void joinAndEnter(Runnable logic)
throws InterruptedException
```

Parameters

logic The `Runnable` object which contains the code to execute.

Throws

InterruptedException When this schedulable is interrupted by `RealtimeThread.interrupt()`[101]

[101] Section 5.3.2.2.2

or `AsynchronouslyInterruptedException.fire()`[102] while waiting for the reference count to go to zero.
IllegalThreadStateException when the caller is a Java thread, or when this method is invoked during finalization of objects in scoped memory and entering this scoped memory area would force deletion of the SO that triggered finalization. This would include the scope containing the SO, and the scope (if any) containing the scope containing the SO.
ThrowBoundaryError Thrown when the JVM needs to propagate an exception allocated in `this` scope to (or through) the memory area of the caller. Storing a reference to that exception would cause an `IllegalAssignmentError`[103], so the JVM cannot be permitted to deliver the exception. The `ThrowBoundaryError`[104] is allocated in the current allocation context and contains information about the exception it replaces.
ScopedCycleException when this invocation would break the single parent rule.
IllegalArgumentException when the caller is a schedulable and `logic` is `null`.

joinAndEnter(java.util.function.Supplier<T>)

Same as `joinAndEnter(Runnable)`[105] except that the executed method is called `get` and an object is returned.

Signature

```
public
T joinAndEnter(java.util.function.Supplier<T> logic)
```

Parameters

logic the object who's get method will be executed.

Returns

a result from the computation.

joinAndEnter(BooleanSupplier)

Same as `joinAndEnter(Runnable)`[106] except that the executed method is called `get` and a `boolean` is returned.

[102] Section 8.4.2.1.2
[103] Section 14.2.3.3
[104] Section 14.2.3.6
[105] Section 11.4.3.11.2
[106] Section 11.4.3.11.2

Signature

`public`
`boolean joinAndEnter(BooleanSupplier logic)`

Parameters

logic the object who's get method will be executed.

Returns

a result from the computation.

joinAndEnter(IntSupplier)

Same as `joinAndEnter(Runnable)`[107] except that the executed method is called `get` and an `int` is returned.

Signature

`public`
`int joinAndEnter(IntSupplier logic)`

Parameters

logic the object who's get method will be executed.

Returns

a result from the computation.

joinAndEnter(LongSupplier)

Same as `joinAndEnter(Runnable)`[108] except that the executed method is called `get` and a `long` is returned.

Signature

`public`
`long joinAndEnter(LongSupplier logic)`

Parameters

logic the object who's get method will be executed.

Returns

a result from the computation.

[107] Section 11.4.3.11.2
[108] Section 11.4.3.11.2

joinAndEnter(DoubleSupplier)

Same as `joinAndEnter(Runnable)`[109] except that the executed method is called `get` and a `double` is returned.

Signature

```
public
double joinAndEnter(DoubleSupplier logic)
```

Parameters

logic the object who's get method will be executed.

Returns

a result from the computation.

joinAndEnter(Runnable, javax.realtime.HighResolutionTime<T>)

In the error-free case, `joinAndEnter` combines `join();enter();` such that no `enter()` from another schedulable can intervene between the two method invocations. The resulting method will wait for the reference count on this `ScopedMemory` to reach zero, or for the current time to reach the designated time, then enter the `ScopedMemory` and execute the `run` method from `logic`.

Since the time is expressed as a `HighResolutionTime`[110], this method is an accurate timer with nanosecond granularity. The actual resolution of the timer and even the quantity it measures depends on the clock associated with `time`. The delay time may be relative or absolute. When relative, then the delay is the amount of time given by `time`, and measured by its associated clock. When absolute, then the delay is until the indicated value is reached by the clock. When the given absolute time is less than or equal to the current value of the clock, the call to `joinAndEnter` behaves effectively like `enter(Runnable)`[111].

Throws `IllegalArgumentException` immediately when `logic` is `null`.

When multiple threads are waiting in `joinAndEnter` family methods for a memory area, at most *one* of them will be released each time the reference count goes to zero.

Note that expiration of `time` may cause control to enter the memory area before its reference count has gone to zero.

[109] Section 11.4.3.11.2
[110] Section 9.4.1.2
[111] Section 11.4.3.11.2

Signature

```
public
void joinAndEnter(Runnable logic,
javax.realtime.HighResolutionTime<T> time)
throws InterruptedException
```

Parameters

logic The `Runnable` object which contains the code to execute.
time The time that bounds the wait.

Throws

InterruptedException When this schedulable is interrupted by `RealtimeThread.interrupt()`[112] or `AsynchronouslyInterruptedException.fire()`[113] while waiting for the reference count to go to zero.
IllegalThreadStateException when the caller is a Java thread, or when this method is invoked during finalization of objects in scoped memory and entering this scoped memory area would force deletion of the SO that triggered finalization. This would include the scope containing the SO, and the scope (if any) containing the scope containing the SO.
ThrowBoundaryError Thrown when the JVM needs to propagate an exception allocated in `this` scope to (or through) the memory area of the caller. Storing a reference to that exception would cause an `IllegalAssignmentError`[114], so the JVM cannot be permitted to deliver the exception. The `ThrowBoundaryError`[115] is allocated in the current allocation context and contains information about the exception it replaces.
ScopedCycleException when the caller is a schedulable and this invocation would break the single parent rule.
IllegalArgumentException when the caller is a schedulable and `time` or `logic` is `null`.
UnsupportedOperationException when the wait operation is not supported using the clock associated with `time`.

joinAndEnter(java.util.function.Supplier<P>, javax.realtime.HighResolution

Same as `joinAndEnter(Runnable, HighResolutionTime)`[116] except that the

[112]Section 5.3.2.2.2
[113]Section 8.4.2.1.2
[114]Section 14.2.3.3
[115]Section 14.2.3.6
[116]Section 11.4.3.11.2

executed method is called `get` and an object is returned.

Signature

```
public
P joinAndEnter(java.util.function.Supplier<P> logic,
javax.realtime.HighResolutionTime<T> time)
```

Parameters

logic the object who's get method will be executed.

Returns

a result from the computation.

joinAndEnter(BooleanSupplier, javax.realtime.HighResolutionTime<T>

Same as `joinAndEnter(Runnable, HighResolutionTime)`[117] except that the executed method is called `get` and a `boolean` is returned.

Signature

```
public
boolean joinAndEnter(BooleanSupplier logic,
javax.realtime.HighResolutionTime<T> time)
```

Parameters

logic the object who's get method will be executed.

Returns

a result from the computation.

joinAndEnter(IntSupplier, javax.realtime.HighResolutionTime<T>)

Same as `joinAndEnter(Runnable, HighResolutionTime)`[118] except that the executed method is called `get` and an `int` is returned.

Signature

```
public
```

[117] Section 11.4.3.11.2
[118] Section 11.4.3.11.2

```
int joinAndEnter(IntSupplier logic,
javax.realtime.HighResolutionTime<T> time)
```

Parameters

logic the object who's get method will be executed.

Returns

a result from the computation.

joinAndEnter(LongSupplier, javax.realtime.HighResolutionTime<T>)

Same as `joinAndEnter(Runnable, HighResolutionTime)`[119] except that the executed method is called `get` and a `long` is returned.

Signature

```
public
long joinAndEnter(LongSupplier logic,
javax.realtime.HighResolutionTime<T> time)
```

Parameters

logic the object who's get method will be executed.

Returns

a result from the computation.

joinAndEnter(DoubleSupplier, javax.realtime.HighResolutionTime<T>)

Same as `joinAndEnter(Runnable, HighResolutionTime)`[120] except that the executed method is called `get` and a `double` is returned.

Signature

```
public
double joinAndEnter(DoubleSupplier logic,
javax.realtime.HighResolutionTime<T> time)
```

Parameters

logic the object who's get method will be executed.

Returns

a result from the computation.

[119] Section 11.4.3.11.2
[120] Section 11.4.3.11.2

getParent

Return a reference to this scopes parent scope (e.g., its parent in the single-parent-rule tree).
Available since RTSJ 2.0

Signature
```
public
javax.realtime.MemoryArea getParent()
```
Returns
a reference to the next outer scoped memory region on the caller's scope stack.
- When there is no outer scoped memory and the primordial parent is heap memory, return a reference to `this`.
- When there is no outer scoped memory and the primordial parent is immortal, or when `this` is unreferenced and unpinned, return `null`

*Problem. The single-parent tree is RTT-independent except for the primordial scope. The type of the primordial scope is RTT-dependent. What should we do about that? When called from a RTT that has entered **`this`**, the above rules make some sense, but what if the caller has not even entered the scope, should we throw an exception? Or just return **`null`**? I think the right solution is to return **`this`** whatever the type of the primordial scope. The app can then know that **`null`** means the scope is not pinned and not referenced, and **`this`** means the parent is either heap or immortal. At that point, the app can learn what it wants to know by just finding what memory area contains the scope object.*

visitScopedChildren(ChildScopeVisitor)

@inheritDoc

Signature
```
public
java.lang.Object visitScopedChildren(ChildScopeVisitor visitor)
```
Throws
IllegalArgumentException when `visitor` is `null`.

newArray(java.lang.Class<T>, int)

Allocate an array of the given type in this memory area. This method may be concurrently used by multiple threads.

Signature

```
public
T[] newArray(java.lang.Class<T> type, int number)
```

Parameters

type @inheritDoc

number @inheritDoc

Throws

IllegalArgumentException @inheritDoc

OutOfMemoryError @inheritDoc

IllegalThreadStateException when the caller is a Java thread.

InaccessibleAreaException when the memory area is not in the schedulable's scope stack.

Returns

@inheritDoc

newInstance(java.lang.Class<T>)

Allocate an object in this memory area. This method may be concurrently used by multiple threads.

Signature

```
public
T newInstance(java.lang.Class<T> type)
throws IllegalAccessException, InstantiationException
```

Parameters

type @inheritDoc

Throws

IllegalAccessException @inheritDoc

IllegalArgumentException @inheritDoc

ExceptionInInitializerError @inheritDoc

OutOfMemoryError @inheritDoc

InstantiationException @inheritDoc

IllegalThreadStateException when the caller is a Java thread.

InaccessibleAreaException when the memory area is not in the schedulable's scope stack.

Returns

@inheritDoc

newInstance(java.lang.reflect.Constructor<T>, T[])

Allocate an object in this memory area. This method may be concurrently used by multiple threads.

Signature

```
public
T newInstance(java.lang.reflect.Constructor<T> c, T[] args)
throws IllegalAccessException, InstantiationException,
InvocationTargetException
```

Parameters

c T@inheritDoc
args @inheritDoc

Throws

IllegalAccessException @inheritDoc
InstantiationException @inheritDoc
OutOfMemoryError @inheritDoc
IllegalArgumentException @inheritDoc
IllegalThreadStateException when the caller is a Java thread.
InvocationTargetException @inheritDoc
InaccessibleAreaException when the memory area is not in the schedulable's scope stack.

Returns

@inheritDoc

setPortal(Object)

Sets the *portal* object of the memory area represented by this instance of `ScopedMemory` to the given object. The object must have been allocated in this `ScopedMemory` instance.

Signature

```
public
void setPortal(Object object)
```

Parameters

object The object which will become the portal for this. When `null` the previous portal object remains the portal object for this or when there was no previous portal object then there is still no portal object for `this`.

Throws

IllegalThreadStateException when the caller is a Java Thread.

IllegalAssignmentError when the caller is a schedulable, and `object` is not allocated in this scoped memory instance and not `null`.

InaccessibleAreaException when the caller is a schedulable, `this` memory area is not in the caller's scope stack and `object` is not `null`.

toString

Returns a user-friendly representation of this ScopedMemory of the form "ScopedMemory#<num>" where <num> is a number that uniquely identifies this scoped memory area.

Signature

```
public
java.lang.String toString()
```

Returns

The string representation

11.4.3.12 SizeEstimator

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. When the instance itself is allocated in several parts (when 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.

See Section MemoryArea.MemoryArea(SizeEstimator)

See Section LTMemory.LTMemory(SizeEstimator)

Inheritance

java.lang.Object
  javax.realtime.SizeEstimator

11.4.3.12.1 Constructors

SizeEstimator

Signature

```
public
  SizeEstimator()
```

11.4.3.12.2 Methods

---

reserve(java.lang.Class, int)

Take into account additional `number` instances of Class `c` when estimating the size of the `MemoryArea`[121].

Signature

```
public
void reserve(java.lang.Class c, int number)
```

Parameters

c The class to take into account.

number The number of instances of `c` to estimate.

Throws

IllegalArgumentException when `c` is `null`.

[121]Section 11.4.3.6

reserve(SizeEstimator, int)

Take into account additional `number` instances of SizeEstimator `size` when estimating the size of the `MemoryArea`[122].

Signature

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

Parameters

estimator The given instance of `SizeEstimator`[123].

number The number of times to reserve the size denoted by `estimator`.

Throws

IllegalArgumentException when `estimator` is `null`.

reserve(SizeEstimator)

Take into account an additional instance of SizeEstimator `size` when estimating the size of the `MemoryArea`[124].

Signature

```
public
void reserve(SizeEstimator size)
```

Parameters

size The given instance of `SizeEstimator`.

Throws

IllegalArgumentException when `size` is `null`.

reserveArray(int)

Take into account an additional instance of an array of `length` reference values when estimating the size of the `MemoryArea`[125].

Available since RTSJ 1.0.1

[122] Section 11.4.3.6
[123] Section 11.4.3.12
[124] Section 11.4.3.6
[125] Section 11.4.3.6

Signature

```
public
void reserveArray(int length)
```

Parameters

length The number of entries in the array.

Throws

IllegalArgumentException when `length` is negative.

reserveArray(int, java.lang.Class)

Take into account an additional instance of an array of `length` primitive values when estimating the size of the `MemoryArea`[126].

Class values for the primitive types are available from the corresponding class types; e.g., Byte.TYPE, Integer.TYPE, and Short.TYPE.

Available since RTSJ 1.0.1

Signature

```
public
void reserveArray(int length, java.lang.Class type)
```

Parameters

length The number of entries in the array.

type 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 when `length` is negative, or `type` does not represent a primitive type.

getEstimate

Gets an estimate of the number of bytes needed to store all the objects reserved.

Signature

```
public
long getEstimate()
```

[126] Section 11.4.3.6

Returns

The estimated size in bytes.

11.4.3.13 StackedMemory

`StackedMemory` implements a scoped memory allocation area and backing store management system. It is designed to allow for safe, fragmentation-free management of scoped allocation with certain strong guarantees provided by the virtual machine and runtime libraries.

Each `StackedMemory` instance represents a single object allocation area and additional memory associated with it in the form of a *backing store*. The backing store associated with a `StackedMemory` is a fixed-size memory area allocated at or before instantiation of the `StackedMemory`. The object allocation area is taken from the associated backing store, and the backing store may be further subdivided into additional `StackedMemory` allocation areas or backing stores by instantiating additional `StackedMemory` objects.

When a `StackedMemory` is created with a backing store, the backing store may be taken from a notional global backing store, in which case it is effectively immortal, or it may be taken from the enclosing `StackedMemory`'s backing store when the scope in which it is created is also a `StackedMemory`, in which case it is returned to its enclosing scope's backing store when the object is finalized. Implementations are not required to return the space occupied by backing stores taken from the global backing store when their associated `StackedMemory` object is finalized.

These backing store semantics divide instances of `StackedMemory` into two categories:

- *Host* — this denotes a `StackedMemory` with an object allocation area created in a new backing store, allocated either from the global store or from a parent `StackedMemory`'s backing store.
- *guest* — this in turn indicates a `StackedMemory` with an object allocation area taken directly from a parent `StackedMemory`'s backing store without creating a sub-store.

In addition, there is one distinguished status for `StackedMemory` objects, *root*. A root `StackedMemory` is a host `StackedMemory` created with a backing store drawn directly from the global backing store, created in an allocation context of some type other than `StackedMemory`.

Allocations from a `StackedMemory` object allocation area are guaranteed to run in time linear in the size of the allocation. All memory for the backing store must be reserved at object construction time.

StackedMemory memory areas have two additional stacking constraints in addition to the single parent rule, designed to enable fragmentation-free manipulation:

- A StackedMemory that is created when another StackedMemory is the current allocation context can only be entered from the same allocation context in which it was created.
- A guest StackedMemory cannot be created from a StackedMemory that currently has another child area that is also a guest StackedMemory. (That is, a StackedMemory can have at most one direct child that is a guest StackedMemory.)

The StackedMemory constructor semantics also enforce the property that a StackedMemory cannot be created from another StackedMemory allocation context unless it is allocated from that context's backing store as either a host or guest area.

The backing store of a StackedMemory behaves as if any StackedMemory object allocation areas are at the "bottom" of the backing store, while the backing stores for enclosed StackedMemory areas are taken from the "top" of the backing store.

There may be an implementation-specific memory overhead for creating a backing store of a given size. This means that creating a StackedMemory with a backing store of exactly the remaining available backing store of the current StackedMemory may fail with an OutOfMemoryError. This overhead must be bounded by a constant.
Available since RTSJ 2.0

Inheritance

java.lang.Object
 javax.realtime.MemoryArea
  javax.realtime.ScopedMemory
   javax.realtime.StackedMemory

11.4.3.13.1 Constructors

StackedMemory(long, long)

Create a host StackedMemory with an object allocation area and backing store of the specified sizes. The backing store is allocated from the currently active memory area when it is also a StackedMemory, and the global backing store otherwise. The object allocation area is allocated from the backing store.

Signature

```
public
  StackedMemory(long scopeSize, long backingSize)
```

Parameters

scopeSize Size of the allocation area

backingSize Size of the total backing store

Throws

IllegalArgumentException when either `scopeSize` or `backingSize` is less than zero, or when `scopeSize` is too large to be allocated from a backing store of size `backingSize`.

OutOfMemoryError when there is insufficient memory available to reserve the requested backing store.

StackedMemory(long, long, Runnable)

Create a host `StackedMemory` with an object allocation area and backing store of the specified sizes, bound to the specified `Runnable`. The backing store is allocated from the currently active memory area when it is also a `StackedMemory`, and the global backing store otherwise. The object allocation area is allocated from the backing store.

Signature

```
public
  StackedMemory(long scopeSize, long backingSize, Runnable logic)
```

Parameters

scopeSize Size of the allocation area

backingSize Size of the total backing store

logic `Runnable` to be entered using `this` as its current memory area when `enter()`[127] is called.

Throws

IllegalArgumentException when either `scopeSize` or `backingSize` is less than zero, or when `scopeSize` is too large to be allocated from a backing store of size `backingSize`.

OutOfMemoryError when there is insufficient memory available to reserve the requested backing store.

[127] Section 11.4.3.13.2

StackedMemory(SizeEstimator, SizeEstimator)

Create a host `StackedMemory` with an object allocation area and backing store of the sizes estimated by the specified `SizeEstimator`s. The backing store is allocated from the currently active memory area when it is also a `StackedMemory`, and the global backing store otherwise. The object allocation area is allocated from the backing store.

Signature

```
public
  StackedMemory(SizeEstimator scopeSize, SizeEstimator backingSize)
```

Parameters

scopeSize `SizeEstimator` indicating the size of the object allocation area

backingSize `SizeEstimator` indicating the size of the total backing store

Throws

IllegalArgumentException when either `scopeSize` or `backingSize` is less than zero, or when `scopeSize` is too large to be allocated from a backing store of size `backingSize`.

OutOfMemoryError when there is insufficient memory available to reserve the requested backing store.

StackedMemory(SizeEstimator, SizeEstimator, Runnable)

Create a host `StackedMemory` with an object allocation area and backing store of the sizes estimated by the specified `SizeEstimator`s, bound to the specified `Runnable`. The backing store is allocated from the currently active memory area when it is also a `StackedMemory`, and the global backing store otherwise. The object allocation area is allocated from the backing store.

Signature

```
public
  StackedMemory(SizeEstimator scopeSize, SizeEstimator backingSize, Runnable lo
```

Parameters

scopeSize `SizeEstimator` indicating the size of the object allocation area

backingSize `SizeEstimator` indicating the size of the total backing store
logic `Runnable` to be entered using `this` as its current memory area when `enter()`[128] is called.

Throws

IllegalArgumentException when either `scopeSize` or `backingSize` is less than zero, or when `scopeSize` is too large to be allocated from a backing store of size `backingSize`.
OutOfMemoryError when there is insufficient memory available to reserve the requested backing store.

StackedMemory(long)

Create a guest `StackedMemory` with an object allocation area of the specified size. The object allocation area is drawn from the parent scope's backing store, which must be a `StackedMemory`.

Signature

```
public
  StackedMemory(long scopeSize)
```

Parameters

scopeSize Size of the allocation area

Throws

IllegalStateException when the parent memory area is not a `StackedMemory`, or when the parent `StackedMemory` already has a child that is also a guest `StackedMemory`.
IllegalArgumentException when `scopeSize` is less than zero.
OutOfMemoryError when there is insufficient memory available in the parent `StackedMemory`'s backing store to reserve the requested object allocation area.

StackedMemory(SizeEstimator)

Create a guest `StackedMemory` with an object allocation area of the size estimated by the specified `SizeEstimator`. The object allocation area is drawn from the parent scope's backing store, which must be a `StackedMemory`.

[128] Section 11.4.3.13.2

Signature

```
public
  StackedMemory(SizeEstimator scopeSize)
```

Parameters

scopeSize `SizeEstimator` indicating the size of the object allocation area

Throws

IllegalStateException when the parent memory area is not a `StackedMemory`, or when the parent `StackedMemory` already has a child that is also a guest `StackedMemory`.

IllegalArgumentException when `scopeSize` is less than zero.

OutOfMemoryError when there is insufficient memory available in the parent `StackedMemory`'s backing store to reserve the requested object allocation area.

StackedMemory(long, Runnable)

Create a guest `StackedMemory` with an object allocation area of the specified size, bound to the specified `Runnable`. The object allocation area is drawn from the parent scope's backing store, which must be a `StackedMemory`.

Signature

```
public
  StackedMemory(long scopeSize, Runnable logic)
```

Parameters

scopeSize Size of the allocation area

logic `Runnable` to be entered using `this` as its current memory area when `enter()`[129] is called.

Throws

IllegalStateException when the parent memory area is not a `StackedMemory`, or when the parent `StackedMemory` already has a child that is also a guest `StackedMemory`.

IllegalArgumentException when `scopeSize` is less than zero.

OutOfMemoryError when there is insufficient memory available in the parent `StackedMemory`'s backing store to reserve the requested object allocation area.

[129] Section 11.4.3.13.2

StackedMemory(SizeEstimator, Runnable)

Create a guest `StackedMemory` with an object allocation area of the size estimated by the specified `SizeEstimator`, bound to the specified `Runnable`. The object allocation area is drawn from the parent scope's backing store, which must be a `StackedMemory`.

Signature

```
public
  StackedMemory(SizeEstimator scopeSize, Runnable logic)
```

Parameters

scopeSize `SizeEstimator` indicating the size of the object allocation area

logic `Runnable` to be entered using `this` as its current memory area when `enter()`[130] is called.

Throws

IllegalStateException when the parent memory area is not a `StackedMemory`, or when the parent `StackedMemory` already has a child that is also a guest `StackedMemory`.

IllegalArgumentException when `scopeSize` is less than zero.

OutOfMemoryError when there is insufficient memory available in the parent `StackedMemory`'s backing store to reserve the requested object allocation area.

11.4.3.13.2 Methods

---

resize(long)

Change the size of the object allocation area for this scope. This method may be used to either grow or shrink the allocation area when there are no objects allocated in the scope and no `Schedulable` object has this area as its current allocation context. It may be used to shrink the allocation area down to the size of its current usage when the calling `Schedulable` object is the only object that has this area on its scope stack and there are no guest `StackedMemory` object allocation areas created after this area in the same backing store but not yet finalized.

[130]Section 11.4.3.13.2

Signature

```
public
void resize(long scopeSize)
```

Parameters

scopeSize The new allocation area size for this scope

Throws

IllegalStateException when the caller is not permitted to perform the requested adjustment or there are additional guest `StackedMemory` allocation areas after this one in the backing store.

OutOfMemoryException when the remaining backing store is insufficient for the requested adjustment.

getMaximumSize

Get the maximum size this memory area can attain. The value returned by this function is the maximum size that can currently be passed to `resize(long)`[131] without triggering an `OutOfMemoryException`.

Signature

```
public
long getMaximumSize()
```

Returns

The maximum size attainable.

enter

Associate this memory area with the current `Schedulable` object for the duration of the `run()` method of the instance of `Runnable` given in this object's constructor. During this period of execution, this memory area becomes the default allocation context until another default allocation context is selected.

This method may only be called from the memory area in which this scope was created.

See Section ScopedMemory.enter()

[131]Section 11.4.3.13.2

Signature

```
public
void enter()
```

Throws

IllegalStateException when the currently active memory area is a `StackedMemory` and is not the area in which this scope was created, or the current memory area is not a `StackedMemory` and this `StackedMemory` is not a root area.
ThrowBoundaryError @inheritDoc
IllegalThreadStateException @inheritDoc
MemoryAccessError @inheritDoc

enter(Runnable)

Associate this memory area with the current `Schedulable` object for the duration of the `run()` method of the given `Runnable`. During this period of execution, this memory area becomes the default allocation context until another default allocation context is selected.

This method may only be called from the memory area in which this scope was created.

See Section ScopedMemory.enter(Runnable)

Signature

```
public
void enter(Runnable logic)
```

Throws

IllegalStateException when the currently active memory area is a `StackedMemory` and is not the area in which this scope was created, or the current memory area is not a `StackedMemory` and this `StackedMemory` is not a root area.
ThrowBoundaryError @inheritDoc
IllegalThreadStateException @inheritDoc
MemoryAccessError @inheritDoc

joinAndEnter

@inheritDoc

Signature

```
public
void joinAndEnter()
```

Throws

IllegalStateException when the currently active memory area is a `StackedMemory` and is not the area in which this scope was created, or the current memory area is not a `StackedMemory` and this `StackedMemory` is not a root area.
InterruptedException @inheritDoc
IllegalThreadStateException @inheritDoc
ThrowBoundaryError @inheritDoc
ScopedCycleException @inheritDoc
MemoryAccessError @inheritDoc

joinAndEnter(javax.realtime.HighResolutionTime<T>)

@inheritDoc

Signature

```
public
void joinAndEnter(javax.realtime.HighResolutionTime<T> time)
```

Throws

IllegalStateException when the currently active memory area is a `StackedMemory` and is not the area in which this scope was created, or the current memory area is not a `StackedMemory` and this `StackedMemory` is not a root area.
InterruptedException @inheritDoc
IllegalThreadStateException @inheritDoc
ThrowBoundaryError @inheritDoc
ScopedCycleException @inheritDoc
MemoryAccessError @inheritDoc

joinAndEnter(Runnable)

@inheritDoc

Signature

```
public
void joinAndEnter(Runnable logic)
```

Throws

IllegalStateException when the currently active memory area is a `StackedMemory` and is not the area in which this scope was created, or the current memory area is not a `StackedMemory` and this `StackedMemory` is not a root area.
InterruptedException @inheritDoc
IllegalThreadStateException @inheritDoc
ThrowBoundaryError @inheritDoc
ScopedCycleException @inheritDoc
MemoryAccessError @inheritDoc

joinAndEnter(Runnable, javax.realtime.HighResolutionTime<T>)

@inheritDoc

Signature

```
public
void joinAndEnter(Runnable logic,
javax.realtime.HighResolutionTime<T> time)
```

Throws

IllegalStateException when the currently active memory area is a `StackedMemory` and is not the area in which this scope was created, or the current memory area is not a `StackedMemory` and this `StackedMemory` is not a root area.
InterruptedException @inheritDoc
IllegalThreadStateException @inheritDoc
ThrowBoundaryError @inheritDoc
ScopedCycleException @inheritDoc
MemoryAccessError @inheritDoc

11.5 The Rationale

11.5.1 The scoped memory model

Languages that employ automatic reclamation of blocks of memory allocated in what is conventionally called the heap by program logic also typically use an algorithm called a garbage collector. Garbage collection algorithms and implementations vary in the amount of non-determinacy they add to the execution of program logic. Rather than require a garbage collector, and require it to meet realtime constraints that would necessarily be a compromise, this specification constructs alternative systems for "safe" management of memory. The scoped and immortal memory areas allow program logic to allocate objects in a Java-like style, ignore the reclamation of those objects, and not incur the latency of the implemented garbage collection algorithm.

The term *scope stack* might mislead a reader to infer that it contains only scoped memory areas. This is incorrect. Although the scope stack may contain scoped memory references, it may also contain heap and immortal memory areas. Also, although the scope stack's behavior is specified as a stack, an implementation is free to use any data structure that preserves the stack semantics.

This specification does not specifically address the lifetime of objects allocated in immortal memory areas. If they were reclaimed while they were still referenced, the referential integrity of the JVM would be compromised which is not permissible. Recovering immortal objects only at the termination of the application, or never recovering them under any circumstances is consistent with this specification.

If a scoped memory area is used by both heap and non-heap SOs, there could be cases where a finalizer executed in non-heap context could attempt to use a heap reference left by a heap-using SO. The code in the finalizer would throw a memory access error. If that exception is not caught in the finalizer, it will be handled by the implementation so finalization will continue undisturbed, but the problem in finalizer that caused the illegal memory access could be hard to locate. So, catch clauses in finalizers for objects allocated in scoped memory are even more useful than they are for normal finalizers.

Support for scoped non-default initial memory areas (SNDIMAs) for schedulables has repercussions. These repercussions include:

- The SNDIMA's parent is set when the SO is constructed, but its reference count is not incremented until the realtime thread is started or the async event handler becomes fireable. This lets a scope with a zero reference count have a parent. This may cause unexpected scoped cycle exceptions. The most surprising are from the `joinAndEnter` family of methods.
- Finalization of a scoped memory (when its reference count goes to zero) can cause finalization of SNDIMAs of AEHs whose AEs are in the finalizing scope.

This can cause finalization of one scope to trigger finalization of numerous other scopes including scopes that are descendants of the scope whose finalization started the process.

- Any action that makes an AEH non-fireable (directly disassociating it from all AEs, or indirectly disassociating it by finalizing the scopes containing those AEs) must block until all all the resulting finalization completes.
- Any action that makes an AEH fireable must block until any ongoing finalization of its SNDIMA completes.

These semantics are complicated (and so prone to bugs in use and in implementation), they may use significant CPU time at unexpected times, CPU time used for this finalization is not controlled by cost enforcement and it is hard to include in feasibility analysis. **Open issue:** This will not encourage the reader! **End of open issue**

Entering the scoped memory when the AEH is fired or the realtime thread starts has almost the same effect as using the scope as an initial memory area with much less complexity that is recommended practice. A future release may deprecate support for SNDIMAs. **Open issue:** *** **End of open issue**

Using heap or immortal memory as the non-default initial memory area of an SO is benign.

11.5.2 The physical memory model

Embedded systems may have many different types of directly addressable memory available to them. Each type has its own characteristics [2] that determine whether it is

- volatile – whether it maintains its state when the power is turned off;
- writable – whether it can be written at all, written once or written many times and whether writing is under program control,
- synchronous or asynchronous – whether the memory is synchronized with the system bus,
- erasable at the byte level – if the memory can be overwritten whether this is done at the byte level or whether whole sectors of the memory need to be erased,
- fast to access – both for reading and writing.

Examples include the following [2]:

- *Dynamic Random Access Memory* (DRAM) and *Static Random Access Memory* (SRAM) – these are volatile memory types that are usually writable at the byte level. There are no limits on the number of times the memory contents can be written. From the embedded systems designer's view point, the main differences between the two are their access times and their costs per byte. SRAM has faster access times and is more expensive. Both DRAM and SRAM are example of asynchronous memory, SDRAM and SSRAM are their synchronized counterparts. Another important difference is that DRAM requires periodic refresh operations, which may interfere with execution time determinism.

- Read-Only Memory (for example, *Erasable Programmable Read-Only Memory* (EPROM)) – these are nonvolatile memory types that once initialized with data can not be overwritten by the program (without recourse to some external effect, usually ultraviolet light as in EPROM). They are fast to access and cost less per byte than DRAM.
- Hybrid Memory (for example, *Electrically Erasable Programmable Read-Only Memory* (EEPROM), and Flash) – these have some properties of both random access and read-only memory.
  - EEPROM – this is nonvolatile memory that is writable at the byte level. However, there are typically limits on how many time the same location can be overwritten. EEPROMs are expensive to manufacture, fast to read but slow to write.
  - FLASH memory – this is nonvolatile that is writable at the sector level. Like EEPROM there are limits on how many times the same location can be overwritten and they are fast to read but slow to write. Flash memory is cheaper to manufacture than EEPROM.

Some embedded systems may have multiple types of random-access memory, and multiple ways of accessing memory. For instance, there may be a small amount of very fast RAM on the processor chip, memory that is on the same board as the processor, memory that may be added and removed from the system dynamically, memory that is accessed across a bus, access to memory that is mediated by a cache, access where the cache is partially disabled so all stores are "write through", memory that is demand paged, and other types of memory and memory-access attributes only limited by physics and the imagination of electrical engineers. Some of these memory types will have no impact on the programmer, others will.

Individual computers are often targeted at a particular application domain. This domain will often dictate the cost and performance requirements, and therefore, the memory type used. Some embedded systems are highly optimized and need to explore different options in memory to meet their performance requirements. Here are five example scenarios.

- Ninety percent of performance-critical memory access is to a set of objects that could fit in a half the total memory.
- The system enables the locking of a small amount of data in the cache, and a small number of pages in the translation lookaside buffer (TLB). A few very frequently accessed objects are to be locked in the cache and a larger number of objects that have jitter requirements can be TLB-locked to avoid TLB faults.

- The boards accept added memory on daughter boards, but that memory is not accessible to DMA from the disk and network controllers and it cannot be used for video buffers. Better performance is obtained if we ensure that all data that might interact with disk, network, or video is not stored on the daughter board.
- Improved video performance can be obtained by using an array as a video buffer. This will only be effective if a physically contiguous, non-pageable, DMA-accessible block of RAM is used for the buffer and all stores forced to write through the cache. Of course, such an approach is dependent on the way the JVM lays out arrays in memory, and it breaks the JVM abstraction by depending on that layout.
- The system has banks of SRAM and saves power by automatically putting them to "sleep" whenever they stay unused for 100 ms or so. To exploit this, the objects used by each phase of our program can be collected in a separate bank of this special memory.

To be clear, few embedded systems are this aggressive in their hardware optimization. The majority of embedded systems have only ROM, RAM, and maybe flash memory. Configuration-controlled memory attributes (such as page locking, and TLB behavior) are more common.

As well as having different types of memory, many computers map input and output devices so that their registers can be accessed as if they were resident within the computer memory (see Section 12.3.1). Hence, some parts of the processor's address space map to real memory and other parts map to device registers. Logically, even a device's memory can be considered part of the memory hierarchy, even where the device's interface is accessed through special assembly instructions. Multiprocessor systems add a further dimension to the problem of memory access. Memory may be local to a CPU, tightly shared between CPUs, or remotely accessible from the CPU (but with a delay).

Traditionally, Java programmers are not concerned with these low-level issues; they program at a higher level of abstraction and assume the JVM makes judicious use of the underlying resources provided by the execution platform[132]. Embedded systems programmers cannot afford this luxury. Consequently, any Java environment that wishes to facilitate the programming of embedded systems must enable the programmer to exercise more control over memory.

[132] This is reflected by the OS support provided. For example, most POSIX systems only offer programs a choice of demand paged or page-locked memory.

11.5.2.1 Problems with the current RTSJ 1.0.2 Physical Memory Framework

The `RTSJ` 1.0.2 supports three ways to allocate objects that can be placed in particular types of memory:

- `ImmortalPhysicalMemory` allocates immortal objects in memory with specified characteristics.
- `LTPhysicalMemory` allocates scoped memory objects in a memory with specified characteristics using a linear time memory allocation algorithm.
- `VTPhysicalMemory` allocates scoped memory objects in memory with specified characteristics using an algorithm that may be worse than linear time but could offer extra services (such as extensibility).

The only difference between the physical memory classes and the corresponding non-physical classes is that the ordinary memory classes give access to normal system RAM and the physical memory classes offer access to particular types of memory.

The `RTSJ` 1.0.2 supports access to physical memory via a memory manager and one or more memory filters. The goal of the memory manager is to provide a single interface with which the programmer can interact in order to access memory with a particular characteristic. A memory filter enables access to a particular type of physical memory. Memory filters may be dynamically added and removed from the system, and there can only be a single filter for each memory type. The memory manager is unaware of the physical addresses of each type of memory. This is encapsulated by the filters. The filters also know the virtual memory characteristics that have been allocated to their memory type. For example, whether the memory is readable or writable.

In theory, any developer can create a new physical memory filter and register it with the PMM. However, the programming of filters is difficult for the following reasons.

- Physical memory type filters include a memory allocation function that must respond to allocation requests with whether a requested range of physical memory is free and if it is not, the physical address of the next free physical memory of the requested type. This is complex because requests for compound types of physical memory must find a free segment that satisfies all attributes of the compound type.
- The Java runtime must continue to behave correctly under the Java memory model when it is using physical memory. This is not a problem when a memory type behaves like the system's normal RAM with respect to the properties addressed by the memory model, or is more restricted than normal RAM (as, for instance, write-through cache is more restricted than copy-back cache). If a new memory type does not obey the memory model using the same instruction sequences as normal RAM, the memory filter must cooperate with the inter-

preter, the JIT, and any ahead-of-time compilation to modify those instruction sequences when accessing the new type of memory. That task is difficult for someone who can easily modify the Java runtime and nearly impossible for anyone else.

Hence, the utility of the physical memory filter framework at Version 1.0.2 is questionable, and hence is replaced in 2.0 with an easier to use framework.

11.5.2.2 The RTSJ 2.0 Physical Memory Framework

The main problem with the 1.0.2 framework is that it places too greater a burden on the JVM implementer. Even for embedded systems, the JVM implementer requires the VM to be portable between systems within the same processor family. It, therefore, cannot have detailed knowledge of the underlying memory architecture. It is only concerned with the standard RAM provided to it by the host operating system.

The design of 2.0 model is based on two constraints:

- Java objects can only be allocated in a memory area if the physical backing store supports the Java Memory Model without the JVM having to perform any operation addition to those that it performs when accessing as the main RAM for the host machine. No extra compiler or JVM interactions shall be required. Hence memory types (such as EEPROM), which potentially require special hardware instructions to perform write operations, cannot be used as the backing store for physical memory areas. Similarly, non-volatile memory can be used any objects store therein may contain references to objects in volatile memory. Although these memory types are prohibited from being used as backing stores, they contain objects of primitive Java types and be accessed via the `RTSJ` Raw Memory facilities (see Section 12.3.1).
- Any API must delegates detailed knowledge of the memory architecture to the programmer of the specific embedded system to be implemented. There is less requirement for portability here, as embedded systems are usually optimized for their host environment. The model assumes that the programmer is aware of the memory map, either through some native operating system interface[133] or from some property file read at program initialization time.

When accessing physical memory, there are two main considerations:

1. the characteristics of the required physical memory, and
2. how that memory is to be mapped into the virtual memory of the application.

[133] For example, the *Advanced Configuration and Power Interface* (ACPI) specification is an open standard for device configuration and power management by the operating system. The ACPI defines platform-independent interfaces for hardware discovery, configuration, power management and monitoring. See http://www.acpi.info/

2.0 requires the program to identify (and inform the RTSJ's physical memory manager of) the physical memory characteristics and the range of physical addresses those characteristic apply to. For example, that there is SRAM between physical address range 0x100000000 and 0xA0000000.

The physical memory manager supports a range of options for mapping physical memory into the virtual memory of the application. Examples of such options are whether the range is to be permanently resident in memory (the default is that it may be subject to paging/swapping), and whether data is written to the cache and the main memory simultaneously (i.e., a write through cache).

Given the required physical and virtual memory characteristics, the programmer requests that the PMM creates a memory filter for accessing this memory. This filter can then be used with new constructors on the physical memory classes. For example,

```
public ImmortalPhysicalMemory(PhysicalMemoryFilter filter, long size)
```

Use of this constructor enables the programmer to specify the allocation of the backing store in a particular type of memory with a particular virtual memory characteristic. The filter is used to locate an area in physical memory with the required physical memory characteristics and to direct its mapping into the virtual address space. Other constructors allow multiple filters to be passed.

Hence, once the filters have been created, physical memory areas can be created and objects can be allocated within those memory areas using the usual RTSJ mechanisms for changing the allocation context of the `new` operator.

11.5.2.3 An example

Consider an example of a system that has a SRAM physical memory module configured at a physical base address of `0x10000000` and of length `0x20000000`. Another module (base address of `0xA0000000` and of length `0x10000000`) also supports SRAM, but this module has been configured so that it saves power by sleeping when not in use. The following subsections illustrate how the embedded programmer informs the PMM about the structure during the program's initialization phase, and how the memory may be subsequently used after this. The example assumes that the PMM supports the virtual memory characteristics defined above.

Program Initialization

For simplicity, the example requires that the address of the memory modules are known, rather than being read from a property file. The program needs to have a class that implements the `PhysicalMemoryCharacteristic`. In this simple example, this is empty.

```
public class MyMemoryType implements PhysicalMemoryCharacteristic {}
```

The initialization method must now create instances of the `PhysicalMemoryModule` class to represent the physical memory module memory modules to represent

```
PhysicalMemoryModule staticRam = new PhysicalMemoryModule(
                                0x10000000L, 0x100000000L);
PhysicalMemoryModule staticSleepableRam = new PhysicalMemoryModule(
                                0xA0000000L, 0x100000000L);
```

It then creates names for the characteristics that the program wants to associate with each memory module.

```
PhysicalMemoryCharacteristic STATIC_RAM = new MyMemoryType();
PhysicalMemoryCharacteristic AUTO_SLEEPABLE = new MyMemoryType();
```

It then informs the PMM of the appropriate associations:

```
NewPhysicalMemoryManager.associate(STATIC_RAM, staticRam);
NewPhysicalMemoryManager.associate(STATIC_RAM,
                                  staticSleepableRam);
NewPhysicalMemoryManager.associate(AUTO_SLEEPABLE,
                                  staticSleepableRam);
```

Once this is done, the program can now create a filter with the required properties. In this case it is for some SRAM that must be auto sleepable.

```
PhysicalMemoryCharacteristic [] PMC =
        new PhysicalMemoryCharacteristic[2];
PMC[0] = STATIC_RAM;
PMC[1] = AUTO_SLEEPABLE;

PhysicalMemoryFilter filter = NewPhysicalMemoryManager.
                             createFilter(PMC, DISABLED, FIXED);
```

If the program had just asked for SRAM then either of the memory modules could satisfy the request.

The initialization is now complete, and the programmer can use the memory for storing objects, as shown below.

Using Physical Memory

Once the programmer has configured the JVM so that it is aware of the physical memory modules, and the programmer names for characteristics of those memory modules, using the physical memory is straight forward. Here is an example.

```
ImmortalPhysicalMemory IM = new ImmortalPhysicalMemory(filter,
                                                        0x1000);
IM.enter(new Runnable() {
    public void run() {
      // The code executing here is running with its allocation
      // context set to a physical immortal memory area that is
      // mapped to RAM which is auto sleepable.
      // Any objects created will be placed in that
      // part of physical memory.
    }
  });
```

It is the appropriate constructor of the physical memory classes that now interfaces with the PMM. The physical memory manager keeps track of previously allocated memory and is able to determine if memory is available with the appropriate characteristics. Of course, the PMM has no knowledge of what these names mean; it is merely providing a look-up service.

Chapter 12

Devices and Triggering

12.1 Overview

Interacting with the external environment in a timely manner is an important requirement for realtime, embedded systems. From an embedded systems' perspective, all interactions with the physical world are performed by input and output devices. Hence, the problem is one of controlling and monitoring of devices. This is an area insufficiently addressed by other Java standards. A conventional Java Virtual Machine is not designed to support device access and interrupt handling. Programs that need this functionality must resort to code written in another language and called via the Java Native Interface (JNI). This specification addresses the problem by providing APIs for interrupt handling and direct memory access without resorting to JNI.

In contrast to earlier versions of this specification, 2.0 has extended the goals of the device interfaces to be type safe and user extensible, so that the user can defined new devices without changing the underlying virtual machine.

There are at least four execution (runtime) environments for the `RTSJ`:

1. on a realtime operating system where the Java application runs in user mode;
2. on a realtime operating system where the Java application runs in a context with a user space device driver;
3. as a "kernel module" incorporated into a realtime kernel where both kernel and application run in supervisor mode; and
4. as part of an embedded device where the Java application runs stand-alone on a hardware machine.

In execution environment 1, interaction with the embedded environment is usually via operating system calls using Java's connection-oriented APIs. The Java program will typically have no direct access to the I/O devices. Although some limited access to physical memory may be provided, it is unlikely that interrupts can be directly handled. However, asynchronous interaction with the environment is still possible, for example, via POSIX signals.

In execution environments 2, 3, and 4, the Java program may be able to directly access devices and handle interrupts.

A device can be anything from a simple set of registers wired to sensors and actuators to a full processor performing some fixed task. The interface to a device is usually through a set of device registers. Depending on the I/O architecture of the processor, the programmer can either access these registers via predetermined memory location (called *memory mapped I/O*) or via special assembler instructions (called *port-mapped I/O*).

A computer system with processing devices can be considered to be a collection of parallel threads. The device 'thread' can communicate and synchronize with the tasks executing inside the main processor either by having the main processor poll registers of the device or via a signal from the device. This signal is usually referred to as an interrupt. All high-level models of device programming must provide [3]

1. facilities for representing, addressing and manipulating device registers; and
2. a suitable representation of interrupts (if interrupts are to be handled).

Version 1.0 of the `RTSJ` went some way towards supporting this model through the notion of *happenings* and the *physical and raw memory* access facilities. Unfortunately, happenings were under defined and the mechanisms for physical and raw memory were overly complex with no clear delineation of the separations of concerns between application developers and JVM implementers.

2.0 has significantly enhanced the support for happenings, and has provided a clearer separation between physical and raw memory. The interfaces for Happening, Timer, and POSIXSignal are now unified under ActiveEvent. This means that both `Happening` and `POSIXSignal`, like `Timer` are now subclasses of AsyncEventHandler. As described in Chapter 8, `ActiveEvent` provides a common light-weight means of notifying that its event has occurred. Unlike `fire()`, where dispatching of the associated handlers is done in context of the caller, an `ActiveEvent` separates this notification that the event occurred, its triggering, from the dispatching by providing its own execution context for the dispatching. As with `Timer`, each class has its own ActiveEventDispatcher: HappeningDispatcher, POSIXSignalDispatcher, and TimeDispatcher. Finally, support for POSIX realtime signals is provided by POSIXRealtimeSignal with its associated dispatcher.

12.2 Definitions

A *happening* is an event that takes place outside the Java runtime environment. The triggers for happenings depend on the external environment, but happenings might include signals and interrupts.

12.3 Semantics

There are several aspects of the API for supporting devices. Raw Memory provides the means of accessing the I/O register of a device. Direct Memory Access (DMA) support provide a means of transferring data using a DMA controller. Active events and dispatchers support releasing event handlers based on external events. Interrupt service routines and application-defined clocks are for linking external events to the active events.

12.3.1 Raw Memory

Raw Memory provides means of accessing particular physical memory addresses as variables of Java's primitive data types, and thereby provides an application with direct access to physical memory, for example, for memory-mapped I/O.

Java objects or references therefore *cannot* be stored in raw memory. The following specifies the `RTSJ`'s facilities for raw memory access.

- Each area of memory supporting raw memory access is identified by a subclass of RawMemoryRegion.
  - The raw memory region RawMemoryFactory.MEMORY_MAPPED_REGION facilitates access to memory 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 memory mapped.
  - The raw memory region RawMemoryFactory.IO_PORT_MAPPED_REGION facilitates access to locations that are outside the main memory used by the JVM. It is used to access input and output device registers when such registers are port-based and can only be accessed by special hardware instructions.
  - The application developer can define and register additional regions to support things like emulated access to devices or access to a bus over a bus controller.
- Access to raw memory is controlled by implementation-defined objects, called *accessor objects*. These implement specification-defined interfaces (e.g., `RawByte`, `RawShort`, `RawInt`, etc.) and are created by implementation-defined factory objects. Each factory implements the `RawMemoryRegionFactory` interface, and is identified by its `RawMemoryRegion`.
- The `RawMemoryFactory` class defines the applications programmers interface to the raw memory facilities.
- The `RawMemoryRegionFactory` interface defines the interface that all factories must support for creating accessor objects.

Figure 12.1: Raw Memory Interface

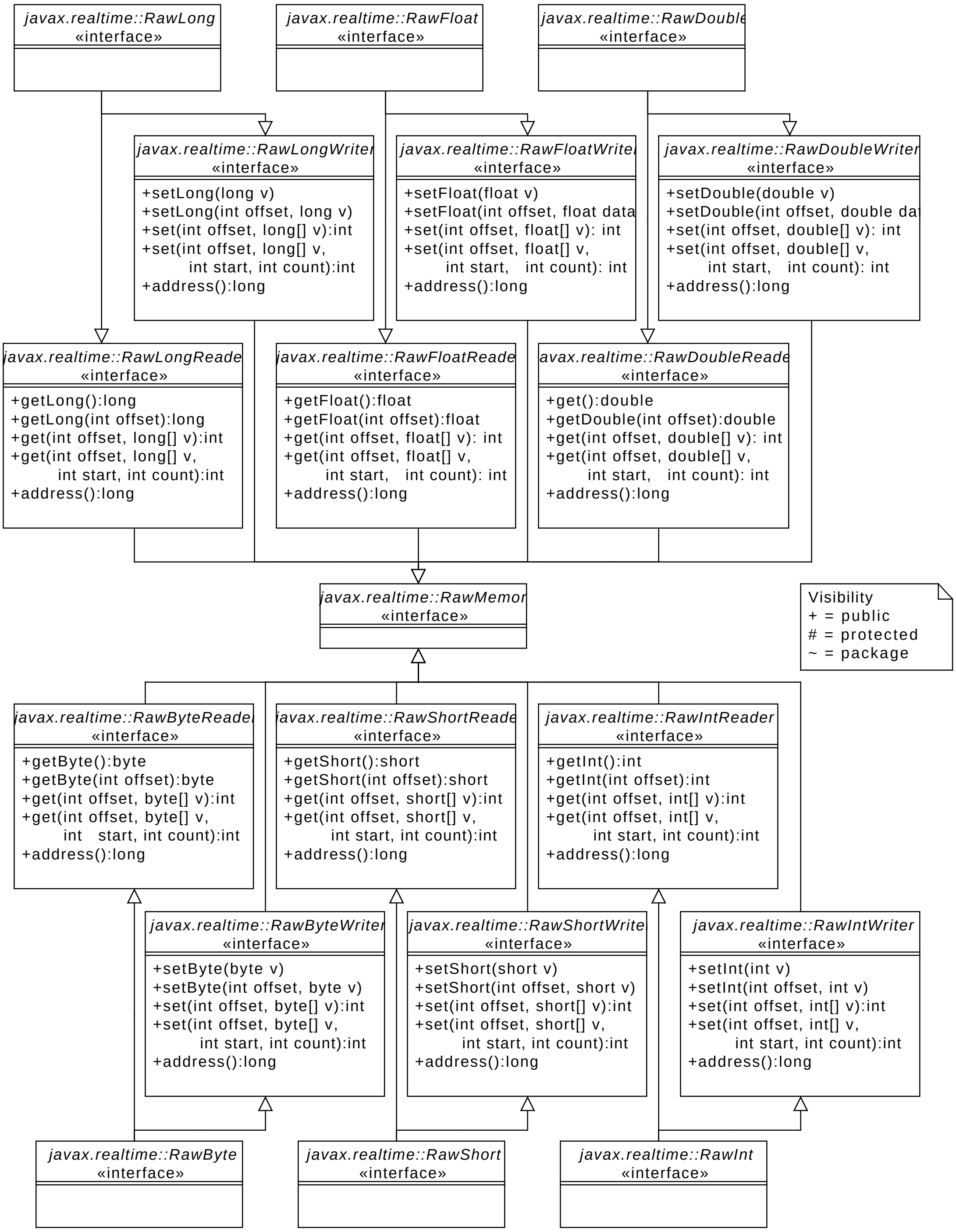
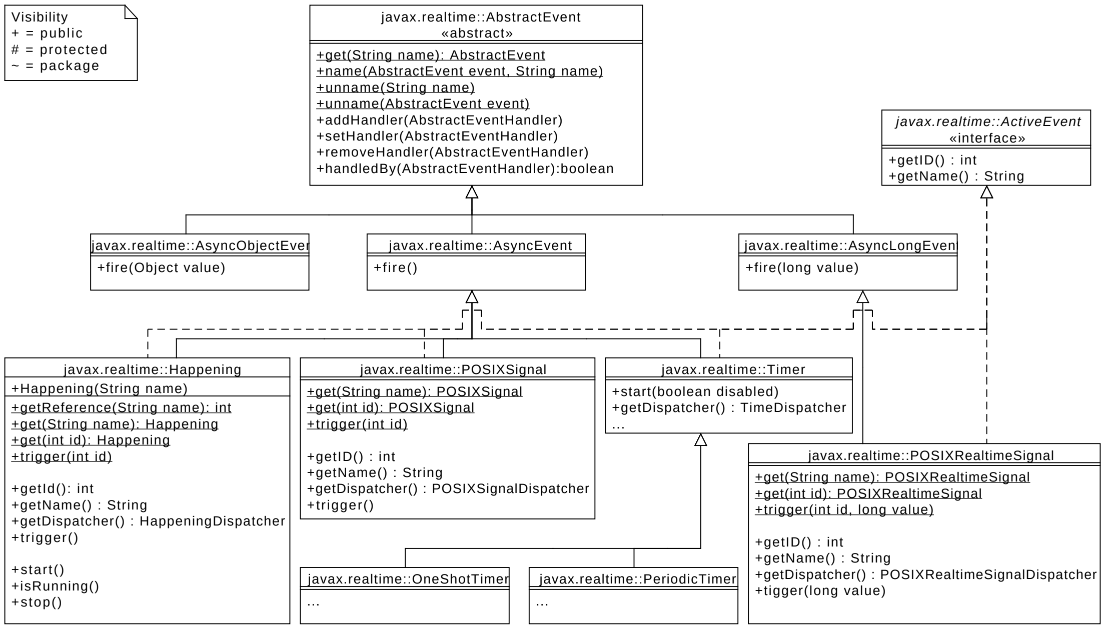


Figure 12.2: Event Classes



12.3.1.1 Raw Memory Region

Raw memory is designed to support arbitrary I/O address spaces. The simplest is through the processor address space and is accessible via standard memory access instructions, such as `load` and `store`. This provide access to memory mapped I/O devices, but there are others address spaces as well. Each of these address spaces is referred to as a *Raw Memory Region*.

There are two raw memory regions that can be supported generically. Memory mapped I/O is one. The other is port mapped I/O. The most common instance is the I/O space provided by Intel x86 compatible processors through their **in** and **out** instructions. The memory mapped I/O raw memory region must be supported by all implementations, but the port mapped I/O raw memory region must only be supported on processors that support it.

All other raw memory regions are optional and may be provided by a system integrator or an application developer. The API provides an interface, RawMemoryRegionFactory, that can be implemented to provide a means of creating accessor objects for that region. These additional regions can be anything from an I/O space provided by a memory mapped device, using memory mapped I/O to implement it, to a purely synthetic I/O space to emulated hardware that has not yet been built.

Each raw memory region is identified by its raw memory region object. These "types" are defined by instances of RawMemoryRegion: RawMemoryFactory.MEMORY_MAPPED_REGION for memory mapped devices and RawMemoryFactory.IO_PORT_MAPPED_REGION for port mapped devices for processors that have instructions for reading and writing an I/O bus directly. The instances are used to get accessors of a region instead of using a RawMemoryRegionFactory directly.

12.3.1.2 Raw Memory Factory

In order to support a variety of device address spaces efficiently, raw memory objects are created using the factory methods provided by RawMemoryFactory. This factory provides static methods to get accessors for a region via a region's type. Regions created during runtime can be provided by registering their factory with the main raw memory factory, so the application code only needs to have a reference to the object identifying the required region. For instance, one could create an I2C raw memory region by implementing a factory for it using a memory mapped I2C controller.

12.3.1.3 Stride

Since the word size of devices do not always match the word size of the memory or I/O bus, the interface provides for the notion of stride. Stride defines the distance between elements in a raw memory area. Normally elements of a memory area are mapped sequentially, without any space between the elements. This is a stride of one. A stride of two, means that every other element in physical memory is mapped into the raw memory area.

For example, it is often easier to map a 16 bit device into a 32 bit system by mapping the 16 bit registers at 32 bit intervals. This enables 16 bit accesses to the device to be atomic on 32 bit addressed systems, even when the bus always does 32 bit transfers. One can create a RawShort area with a stride of two. Then the area can be accessed as if the registers where contiguous.

Since stride is designed to support mapping devices that have a smaller word size than the host machine, the implementation is allowed to assume that the padding between values is "do not care" data, and can be overwritten arbitrarily.

12.3.2 Direct Memory Access Support

Many embedded systems provide a means of moving data without direct involvement of the main processor. This is typically programmed with a special device called a DMA controller. DMA controllers are treated specially since they are central to bulk transfer in device drivers. The data to be transferred is not in device registers, but in normal RAM. Java already provides an API form managing this kind of memory in `java.nio`. The DMA API defined here provides a seamless means of integrating those features into a device driver for DMA.

There are various architectures for DMA controllers, each requiring its own programming paradigm, so only common low level support is provided by this specification. Raw memory can be used to program the DMA controller, but there needs to be a means of representing bulk data. The `java.nio.ByteBuffer` provides just such a representation. The only difference is that the restrictions on the memory behind byte buffer objects is a bit different than for other `java.nio` mechanisms.

These differences are covered with a special byte buffer factory: RawBufferFactory. An instance of this factory can produce direct byte buffers within a given memory range. This range can be chosen by the programmer to be within the range of a given DMA controller. The factory also provides methods for getting the start address of a buffer's memory and checking if a buffer's memory is within a given range. These addresses should be compatible with DMA controllers in the system, though for controllers with a smaller address space than the processor, the DMA address may have fixed offset from the processor physical address. The RawBufferFactory class also provides static methods for ensuring that Java-generated changes to DMA-mapped memory buffers are visible to native code, and vice-versa.

12.3.3 External Triggering

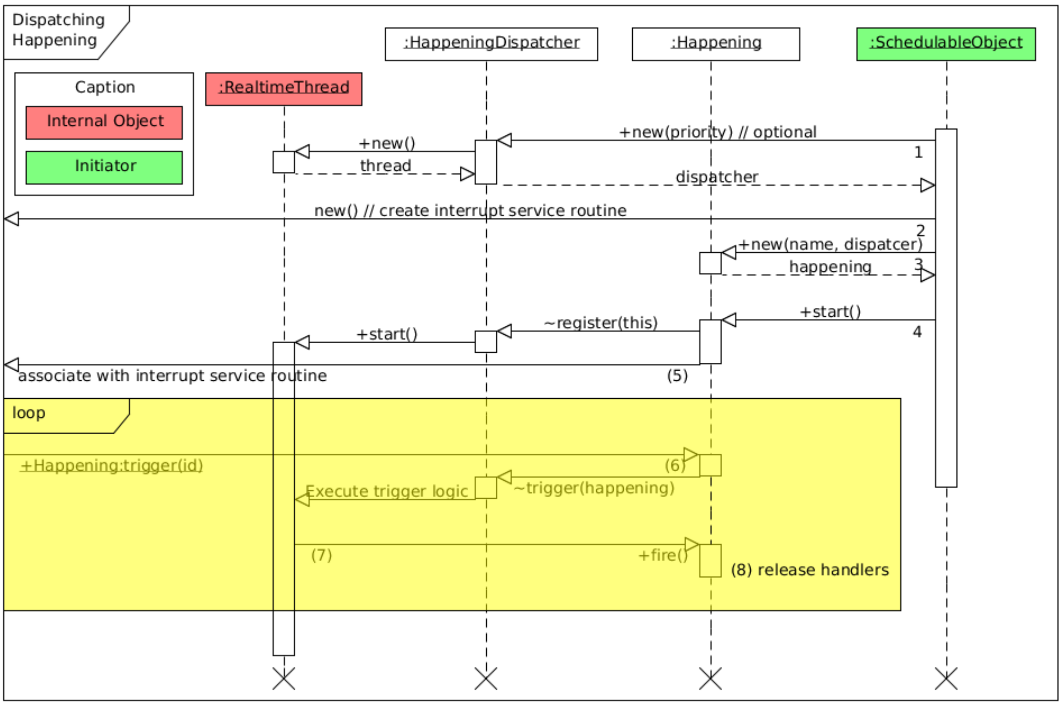
It is not enough to be able to read from and write to devices; many applications, need a means of being interrupted when an event happens. This specification provides a two-level interrupt mechanism. For predefined interfaces, such as POSIX signals, the first level handler is provided by the virtual machine and asynchronous events provide the second level event handling. For external events and additional clocks, where the programmer needs to be able to define new instances and provide for their triggering, additional classes are provided to manage both the first level, as well as the second level handling. In all cases, the user can control the priority and affinity of the dispatching between the first level and second level handing.

12.3.3.1 Happenings

Whereas, in previous versions of this specification, happenings were represented as a `String`, as of 2.0 they have become an object in their own right. This makes it easier to properly type methods that use them and for the user to define new happening for an application without the need to change the JVM. Furthermore, indirection is minimized by making the new `Happening` class a subclass of `AsyncEvent`.

Since a `Happening` needs to be triggerrable from an external event, such as an interrupt, the `Happening` class also implements `ActiveEvent`. As with other active events, `Happening` has its own dispatcher class: `HappeningDispatcher`. There is a

Figure 12.3: Happening State Transition Diagram



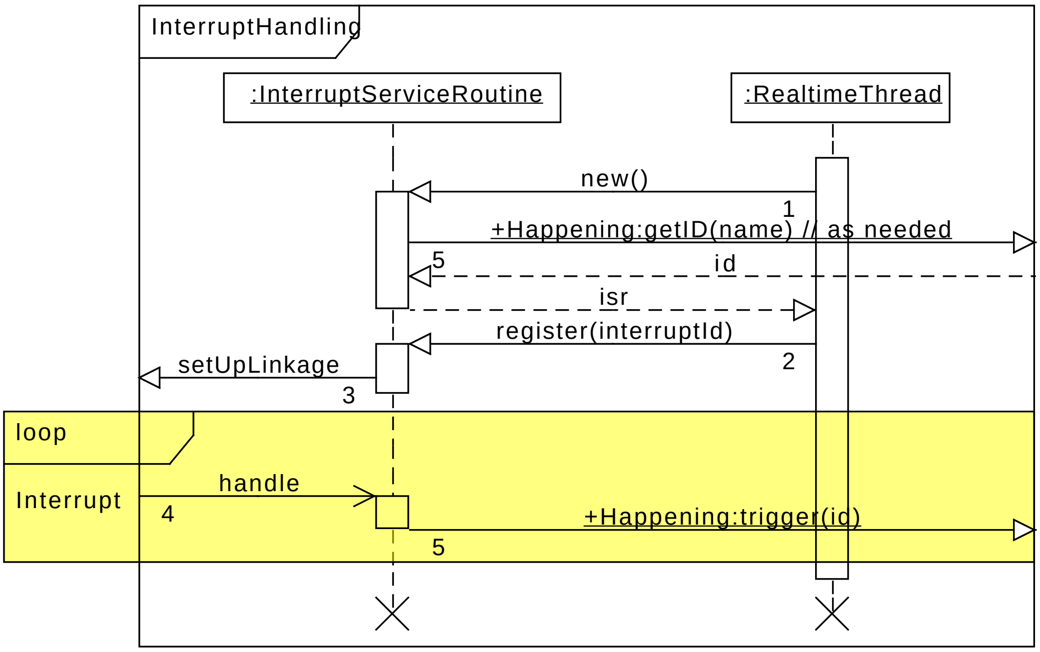
default happening dispatcher that is used when none is provided at creation time, otherwise, the programmer can provide one to change the priority and affinity of dispatching.

Normally, happenings are triggered either from an `InterruptServiceRoutine` or from JNI code. For the later, the interface provides a means of linking a happening by name. This enables native code to get a handle for triggering a happening without have a direct reference. The given name must follow the Java naming conventions. A happening name defined outside of this specification should not begin with `java.` or `javax.`.

Figure 12.3 illustrates the sequence of actions necessary for defining and using a `Happening`. When using an application defined dispatcher, it must be created first (1). When using an `InterruptServiceRoutine` to trigger the happening, it may be created before (2) or after the happening is create. After creating the happening (3), the happening must be started to be registered with it dispatcher to be triggered from native code. Of course, the JVM must have direct access to an interrupt, either by being directly bound in the kernel or by some other means, such as a system call, for setting up user-space device drivers. Only after both an

InterruptServiceRoutine is registered and a Happening with the same name is started, can that happening be triggered (6–8).

Figure 12.4: Interrupt servicing



There are three main differences between this mechanism and the string based API.

- The Happening class is now a first-class entity, rather than being buried in the implementation and identified only by a String object.
- They include the trigger(int) method that enables a happening to be explicitly triggered by Java code, and at the implementation's option, a native code function that permits native application code to trigger the happening.
- Finally, Happening is a subclass of AsyncEvent just as with Timer instead of having a happening attached to an AsyncEvent.

12.3.4 Interrupt Service Routines

In Java-based systems, JNI is typically used to transfer control between the assembler/C *interrupt service routine* (ISR) and the program. 2.0 of the RTSJ supports the possibility of the ISR containing Java code. This is clearly an area where it is difficult to maintain the portability goal of Java. Furthermore, not all `RTSJ` deployments can support `InterruptServiceRoutine`. A JVM that runs in user space does not generally have access to interrupts.

The JVM must either be standalone, running in a kernel module, or running in a special I/O partition on a partitioning OS where interrupts are passed through using some virtualization technique. Hence, JVM support for ISR is not required for `RTSJ` compliance.

Interrupt handling is necessarily machine dependent. However, the `RTSJ` provides an abstract model that can be implemented on top of all architectures.

The following semantic model shall be supported by the `RTSJ`:

- An *occurrence* of an interrupt consists of its *generation* and *delivery*.
- Generation of the interrupt is the mechanism in the underlying hardware or system that makes the interrupt available to the Java program.
- Delivery is the action that invokes an interrupt service routine (ISR) in response to the occurrence of the interrupt. This may be performed by the JVM or application native code linked with the JVM, or directly by the hardware interrupt mechanism.
- Between generation and delivery, the interrupt is *pending*.
- Some or all interrupt occurrences may be inhibited. While an interrupt occurrence is inhibited, all occurrences of that interrupt shall be prevented from being delivered. Whether such occurrences remain pending or are lost is implementation defined, but it is expected that the implementation shall make a best effort to avoid losing pending interrupts.
- Certain implementation-defined interrupts are *reserved*. Reserved interrupts are either interrupts for which application-defined ISRs are not supported, or those that already have ISRs by some other implementation-defined means. For example, a clock interrupt, which is used for internal time keeping by the JVM, is a reserved interrupt.
- An application-defined ISR can be registered with one or more non-reserved interrupts. Registering an ISR for an interrupt shall implicitly deregister any already registered ISR for that interrupt. Any daisy-chaining of interrupt handlers shall be performed explicitly by the application interrupt handlers.
- While an ISR is registered to an interrupt, the `handle` method shall be called *once* for each delivery of that interrupt. For locking out further interrupts during interrupt handling, the `handle` method must be synchronized with a priority high enough to lock out the requisite interrupts. This synchronized uses priority ceiling emulation to inhibit the corresponding interrupt (and all lower priority interrupts). The default allocation context of the `handle` method is the memory area passed during construction.

Any exception propagated from the `handle` method shall be caught by the JVM and ignored.

- Code running in the context of an ISR may only attempt to acquire a lock that has priority ceiling emulation as its monitor control policy. The behavior is undefined, when an ISR attempt to acquire a lock that has a monitor control policy other than priority ceiling emulation. **Open issue:** jjh: Do we want to support a mechanism of static checking here? **End of open issue**

The model assumes that

- the processor has a (logical) interrupt controller that monitors a number of *interrupt lines*;
- the interrupt controller may associate each interrupt line with a particular interrupt priority;
- associated with the interrupt lines is a (logical) interrupt vector that contains the addresses of the ISRs;
- the processor has instructions that allow interrupts from a particular line to be disabled/masked irrespective of whether (or the type of) device attached;
- disabling interrupts from a specific line may disables the interrupts from lines of lower priority;
- a device can be connected to an arbitrary interrupt line;
- when an interrupt is signalled on an interrupt line by a device, the processor uses the identity of the interrupt line to index into the interrupt vector and jumps to the address of the ISR; the hardware automatically disables further interrupts (either of the same priority and lower or, possibly, all interrupts);
- on return from the ISR, interrupts are automatically re-enabled.

For each of the interrupt, the `RTSJ` has an associated hardware priority that can be used to set the ceiling of an ISR object. The `RTSJ` virtual machine may use this to disable the interrupts from the associated interrupt line and lower priority interrupts, when it is executing a synchronized method of the interrupt-handling object. On a multicore system, the situation is more complex, since there may be other cores available to handle other interrupts, even at lower priorities, and some other locking mechanism may be necessary as well.

Though synchronization is not required in general, it is required to enforce visibility of changes made to any variables shared between some normal `Schedulable` and a `handle` method. For the `handle` method, this may be done automatically by the hardware interrupt handling mechanism or it may require added support from the realtime Java virtual machine. However, for clarity of the model, `RTSJ` recommends that the `handle` method should be defined as synchronized.

Support for interrupt handling is encapsulated in the `InterruptServiceRoutine` abstract class that has two main methods. The first is the final `register` method that will register an instance of the class with the system so that the appropriate interrupt vector can be initialized. The second is the abstract `handle` method that provides the code to be executed in response to the interrupt occurring. An individual real-time JVM may place restrictions of the code that can be written in this method. The process is illustrated in Figure 12.4, and is described below.

1. The ISR is created by some application real-time thread.
2. The created ISR is registered with the JVM, the interrupt id is passed as a parameter.
3. As part of the registration process, some internal interface is used to set up the code that will set the underlying interrupt vectors to some C/assembler code that will provide the necessary linkage to allow the callback to the Java handler.
4. When the interrupt occurs, the handler is called.

In order to integrate further the interrupt handling with the Java application, the handle method may trigger a happening or fire an event.

Typically an implementation of the `RTSJ` that supports first-level interrupt handling will document the following items:

1. For each interrupt, its identifying integer value, the priority at which the interrupt occurs and whether it can be inhibited or not, and the effects of registering ISRs to non inhibitable interrupts (if this is permitted).
2. Which run-time stack the `handle` method uses when it executes.
3. Any implementation- or hardware-specific activity that happens before the `handle` method is invoked (e.g., reading device registers, acknowledging devices).
4. The state (inhibited/uninhibited) of the nonreserved interrupts when the program starts; if some interrupts are uninhibited, what the mechanism is that a program can use to protect itself before it can register the corresponding ISR.
5. The treatment of interrupt occurrences that are generated while the interrupt is inhibited; i.e., whether one or more occurrences are held for later delivery, or all are lost.
6. Whether predefined or implementation-defined exceptions are raised as a result of the occurrence of any interrupt (for example, a hardware trap resulting from a segmentation error), and the mapping between the interrupt and the predefined exceptions.
7. On a multiprocessor, the rules governing the delivery of an interrupt occurrence to a particular processor. For example, whether execution of the `handle` method may spin if the lock of the associated object is held by another processor.

12.4 Package javax.realtime.device

12.4.1 Interfaces

12.4.1.1 RawByte

A marker for an object that can be used to access to a single byte. Read and write access to that byte is checked by the factory that creates the instance; therefore, no access checking is provided by this interface, only bounds checking.

Available since RTSJ 2.0

Interfaces
RawByteReader
RawByteWriter

12.4.1.2 RawByteReader

A marker for a byte accessor object encapsulating the protocol for reading bytes from raw memory. A byte accessor can always access at least one byte. Each byte is transfered in a single atomic operation. Groups of bytes may be transfered together; however, this is not required.

Objects of this type are created with the method `RawMemoryFactory.createRawByteReader`[1] and `RawMemoryFactory.createRawByte`[2]. Each object references a range of elements in the `RawMemoryRegion`[3] starting at the *base address* provided to the factory method. The size provided to the factor method determines the number of elements accessable.

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

Available since RTSJ 2.0

Interfaces
RawMemory

---

[1]Section 12.4.3.5.3
[2]Section 12.4.3.5.3
[3]Section 12.4.3.6

12.4.1.2.1 Methods

---

getByte

Get the value at the first position referenced by this instance, i.e., the value at its start address. This operation must be atomic with respect to all other raw memory accesses to the address.

Signature

```
public
byte getByte()
```

Returns

the value at the *base address* provided to the factory method that created this object.

getByte(int)

Get the value of the nth element referenced by this instance, where `n` is `offset` and the address is *base address* + (`offset` ∗ stride ∗ *element size in bytes*). When an exception is thrown, no data is transfered.

Signature

```
public
byte getByte(int offset)
throws OffsetOutOfBoundsException
```

Parameters

offset of byte in the memory region starting from the address specified in the associated factory method.

Throws

OffsetOutOfBoundsException when `offset` is negative or greater than or equal to the number of elements in the raw memory region.

Returns

the value at the address specified.

get(int, byte[])

Fill the `values` starting at the address referenced by this instance plus the offset scaled by the element size in bytes and the object's `stride`. Only the bytes in the intersection of the start and end of `values` and the *base address* and the end of the memory region are transfered. When an exception is thrown, no data is transfered.

Signature

```
public
int get(int offset, byte[] values)
throws OffsetOutOfBoundsException, NullPointerException
```

Parameters

offset of the first byte in the memory region to transfere
values the array to received the bytes

Throws

OffsetOutOfBoundsException when `offset` is negative or greater than or equal to the number of elements in the raw memory region.
NullPointerException when `values` is null.

Returns

the number of elements copied to `values`

get(int, byte[], int, int)

Fill `values` with data from the memory region, where `offset` is first byte in the memory region and `start` is the first index in `values`. The number of bytes transfered is the minimum of `count`, the *size* of the memory region minus `offset`, and length of `values` minus `start`. When an exception is thrown, no data is transfered.

Signature

```
public
int get(int offset, byte[] values, int start, int count)
throws OffsetOutOfBoundsException,
ArrayIndexOutOfBoundsException, NullPointerException
```

Parameters

offset of the first byte in the memory region to transfere
values the array to received the bytes
start the first index in array to fill
count the maximum number of bytes to copy

Throws

OffsetOutOfBoundsException when `offset` is negative or either `offset` or `offset` + `count` is greater than or equal to the size of this raw memory area.
ArrayIndexOutOfBoundsException when `start` is negative or either `start` or `start` + `count` is greater than or equal to the size of values.
NullPointerException when `values` is null or `count` is negative.

Returns

the number of bytes actually transfered.

12.4.1.3 RawByteWriter

A marker for a byte accessor object encapsulating the protocol for writing bytes from raw memory. A byte accessor can always access at least one byte. Each byte is transfered in a single atomic operation. Groups of bytes may be transfered together; however, this is not required.

Objects of this type are created with the method `RawMemoryFactory.createRawByteWriter`[4] and `RawMemoryFactory.createRawByte`[5]. Each object references a range of elements in the `RawMemoryRegion`[6] starting at the *base address* provided to the factory method. The size provided to the factor method determines the number of elements accessable.

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

Available since RTSJ 2.0

Interfaces

RawMemory

12.4.1.3.1 Methods

setByte(byte)

[4]Section 12.4.3.5.3
[5]Section 12.4.3.5.3
[6]Section 12.4.3.6

Set the value at the first position referenced by this instance, i.e., the value at its start address. This operation must be atomic with respect to all other raw memory accesses to the address. When an exception is thrown, no data is transfered.

Signature

```
public
void setByte(byte value)
```

setByte(int, byte)

Set the value of the nth element referenced by this instance, where `n` is `offset` and the address is *base address* + `offset` ∗ *size of Byte*. This operation must be atomic with respect to all other raw memory accesses to the address. When an exception is thrown, no data is transfered.

Signature

```
public
void setByte(int offset, byte value)
throws OffsetOutOfBoundsException
```

Parameters

offset of byte in the memory region.

Throws

OffsetOutOfBoundsException when `offset` is negative or greater than or equal to the number of elements in the raw memory region.

set(int, byte[])

Copy `values` to the raw memory starting at the address referenced by this instance plus the `offset` scaled by the element size in bytes and the objects `stride`. Only the bytes in the intersection of `values` and the end of the memory region are transfered. When an exception is thrown, no data is transfered.

Signature

```
public
int set(int offset, byte[] values)
throws OffsetOutOfBoundsException, NullPointerException
```

Throws

OffsetOutOfBoundsException when `offset` is negative or greater than or equal to the number of elements in the raw memory region.
NullPointerException when `values` is null.

Returns
the number of elements copied to `values`

set(int, byte[], int, int)

Copy `values` to the memory region, where `offset` is first byte in the memory region to write and `start` is the first index in `values` from which to read. The number of bytes transfered is the minimum of `count`, the *size* of the memory region minus `offset`, and length of `values` minus `start`. When an exception is thrown, no data is transfered.

Signature
```
public
int set(int offset, byte[] values, int start, int count)
throws OffsetOutOfBoundsException,
ArrayIndexOutOfBoundsException, NullPointerException
```
Parameters
offset of the first byte in the memory region to set
values the array from which to retrieve the bytes
start the first index in array to fill
count the maximum number of bytes to copy

Throws
OffsetOutOfBoundsException when `offset` is negative or either `offset` or `offset` + `count` is greater than or equal to the size of this raw memory area.
ArrayIndexOutOfBoundsException when `start` is negative or either `start` or `start` + `count` is greater than or equal to the size of values.
NullPointerException when `values` is null.

Returns
the number of bytes actually transfered.

12.4.1.4 RawDouble

A marker for an object that can be used to access to a single double. Read and write access to that double is checked by the factory that creates the instance; therefore, no access checking is provided by this interface, only bounds checking.

Available since RTSJ 2.0

Interfaces
RawDoubleReader
RawDoubleWriter

12.4.1.5 RawDoubleReader

A marker for a double accessor object encapsulating the protocol for reading doubles from raw memory. A double accessor can always access at least one double. Each double is transfered in a single atomic operation. Groups of doubles may be transfered together; however, this is not required.

Objects of this type are created with the method `RawMemoryFactory.createRawDoubleReader`[7] and `RawMemoryFactory.createRawDouble`[8]. Each object references a range of elements in the `RawMemoryRegion`[9] starting at the *base address* provided to the factory method. The size provided to the factor method determines the number of elements accessable.

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

Available since RTSJ 2.0

Interfaces
RawMemory

12.4.1.5.1 Methods

getDouble

Get the value at the first position referenced by this instance, i.e., the value at its start address. This operation must be atomic with respect to all other raw memory accesses to the address.

[7]Section 12.4.3.5.3
[8]Section 12.4.3.5.3
[9]Section 12.4.3.6

Signature

```
public
double getDouble()
```

Returns

the value at the *base address* provided to the factory method that created this object.

getDouble(int)

Get the value of the n^{th} element referenced by this instance, where `n` is `offset` and the address is *base address* + (`offset` ∗ stride ∗ *element size in bytes*). When an exception is thrown, no data is transfered.

Signature

```
public
double getDouble(int offset)
throws OffsetOutOfBoundsException
```

Parameters

offset of double in the memory region starting from the address specified in the associated factory method.

Throws

OffsetOutOfBoundsException when `offset` is negative or greater than or equal to the number of elements in the raw memory region.

Returns

the value at the address specified.

get(int, double[])

Fill the `values` starting at the address referenced by this instance plus the offset scaled by the element size in bytes and the object's `stride`. Only the doubles in the intersection of the start and end of `values` and the *base address* and the end of the memory region are transfered. When an exception is thrown, no data is transfered.

Signature

```
public
int get(int offset, double[] values)
```

`throws OffsetOutOfBoundsException, NullPointerException`

Parameters

offset of the first double in the memory region to transfere
values the array to received the doubles

Throws

OffsetOutOfBoundsException when `offset` is negative or greater than or equal to the number of elements in the raw memory region.
NullPointerException when `values` is null.

Returns

the number of elements copied to `values`

get(int, double[], int, int)

Fill `values` with data from the memory region, where `offset` is first double in the memory region and `start` is the first index in `values`. The number of bytes transfered is the minimum of `count`, the *size* of the memory region minus `offset`, and length of `values` minus `start`. When an exception is thrown, no data is transfered.

Signature

```
public
int get(int offset, double[] values, int start, int count)
throws OffsetOutOfBoundsException,
ArrayIndexOutOfBoundsException, NullPointerException
```

Parameters

offset of the first double in the memory region to transfere
values the array to received the doubles
start the first index in array to fill
count the maximum number of doubles to copy

Throws

OffsetOutOfBoundsException when `offset` is negative or either `offset` or `offset` + `count` is greater than or equal to the size of this raw memory area.
ArrayIndexOutOfBoundsException when `start` is negative or either `start` or `start` + `count` is greater than or equal to the size of values.
NullPointerException when `values` is null or `count` is negative.

Returns

the number of doubles actually transfered.

12.4.1.6 RawDoubleWriter

A marker for a double accessor object encapsulating the protocol for writing doubles from raw memory. A double accessor can always access at least one double. Each double is transfered in a single atomic operation. Groups of doubles may be transfered together; however, this is not required.

Objects of this type are created with the method `RawMemoryFactory.createRawDoubleWriter`[10] and `RawMemoryFactory.createRawDouble`[11]. Each object references a range of elements in the `RawMemoryRegion`[12] starting at the *base address* provided to the factory method. The size provided to the factor method determines the number of elements accessable.

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

Available since RTSJ 2.0

Interfaces
RawMemory

12.4.1.6.1 Methods

setDouble(double)

Set the value at the first position referenced by this instance, i.e., the value at its start address. This operation must be atomic with respect to all other raw memory accesses to the address. When an exception is thrown, no data is transfered.

Signature

```
public
void setDouble(double value)
```

[10] Section 12.4.3.5.3
[11] Section 12.4.3.5.3
[12] Section 12.4.3.6

setDouble(int, double)

Set the value of the n^{th} element referenced by this instance, where `n` is `offset` and the address is *base address* + `offset` ∗ *size of Double*. This operation must be atomic with respect to all other raw memory accesses to the address. When an exception is thrown, no data is transfered.

Signature

```
public
void setDouble(int offset, double value)
throws OffsetOutOfBoundsException
```

Parameters

offset of double in the memory region.

Throws

OffsetOutOfBoundsException when `offset` is negative or greater than or equal to the number of elements in the raw memory region.

set(int, double[])

Copy `values` to the raw memory starting at the address referenced by this instance plus the `offset` scaled by the element size in bytes and the objects `stride`. Only the doubles in the intersection of `values` and the end of the memory region are transfered. When an exception is thrown, no data is transfered.

Signature

```
public
int set(int offset, double[] values)
throws OffsetOutOfBoundsException, NullPointerException
```

Throws

OffsetOutOfBoundsException when `offset` is negative or greater than or equal to the number of elements in the raw memory region.

NullPointerException when `values` is null.

Returns

the number of elements copied to `values`

set(int, double[], int, int)

Copy `values` to the memory region, where `offset` is first double in the memory region to write and `start` is the first index in `values` from which to read. The number of bytes transfered is the minimum of `count`, the *size* of the memory region minus `offset`, and length of `values` minus `start`. When an exception is thrown, no data is transfered.

Signature

```
public
int set(int offset, double[] values, int start, int count)
throws OffsetOutOfBoundsException,
ArrayIndexOutOfBoundsException, NullPointerException
```

Parameters

offset of the first double in the memory region to set
values the array from which to retrieve the doubles
start the first index in array to fill
count the maximum number of doubles to copy

Throws

OffsetOutOfBoundsException when `offset` is negative or either `offset` or `offset` + `count` is greater than or equal to the size of this raw memory area.
ArrayIndexOutOfBoundsException when `start` is negative or either `start` or `start` + `count` is greater than or equal to the size of values.
NullPointerException when `values` is null.

Returns

the number of doubles actually transfered.

12.4.1.7 RawFloat

A marker for an object that can be used to access to a single float. Read and write access to that float is checked by the factory that creates the instance; therefore, no access checking is provided by this interface, only bounds checking.

Available since RTSJ 2.0

Interfaces

RawFloatReader
RawFloatWriter

12.4.1.8 RawFloatReader

A marker for a float accessor object encapsulating the protocol for reading floats from raw memory. A float accessor can always access at least one float. Each float is transfered in a single atomic operation. Groups of floats may be transfered together; however, this is not required.

Objects of this type are created with the method `RawMemoryFactory.createRawFloatReader`[13] and `RawMemoryFactory.createRawFloat`[14]. Each object references a range of elements in the `RawMemoryRegion`[15] starting at the *base address* provided to the factory method. The size provided to the factor method determines the number of elements accessable.

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

Available since RTSJ 2.0

Interfaces

RawMemory

12.4.1.8.1 Methods

getFloat

Get the value at the first position referenced by this instance, i.e., the value at its start address. This operation must be atomic with respect to all other raw memory accesses to the address.

Signature

```
public
float getFloat()
```

Returns

[13]Section 12.4.3.5.3
[14]Section 12.4.3.5.3
[15]Section 12.4.3.6

the value at the *base address* provided to the factory method that created this object.

getFloat(int)

Get the value of the nth element referenced by this instance, where `n` is `offset` and the address is *base address* + (`offset` ∗ stride ∗ *element size in bytes*). When an exception is thrown, no data is transfered.

Signature

```
public
float getFloat(int offset)
throws OffsetOutOfBoundsException
```

Parameters

offset of float in the memory region starting from the address specified in the associated factory method.

Throws

OffsetOutOfBoundsException when `offset` is negative or greater than or equal to the number of elements in the raw memory region.

Returns

the value at the address specified.

get(int, float[])

Fill the `values` starting at the address referenced by this instance plus the offset scaled by the element size in bytes and the object's `stride`. Only the floats in the intersection of the start and end of `values` and the *base address* and the end of the memory region are transfered. When an exception is thrown, no data is transfered.

Signature

```
public
int get(int offset, float[] values)
throws OffsetOutOfBoundsException, NullPointerException
```

Parameters

offset of the first float in the memory region to transfere
values the array to received the floats

Throws

OffsetOutOfBoundsException when `offset` is negative or greater than or equal to the number of elements in the raw memory region.
NullPointerException when `values` is null.

Returns
the number of elements copied to `values`

get(int, float[], int, int)

Fill `values` with data from the memory region, where `offset` is first float in the memory region and `start` is the first index in `values`. The number of bytes transfered is the minimum of `count`, the *size* of the memory region minus `offset`, and length of `values` minus `start`. When an exception is thrown, no data is transfered.

Signature
```
public
int get(int offset, float[] values, int start, int count)
throws OffsetOutOfBoundsException,
ArrayIndexOutOfBoundsException, NullPointerException
```
Parameters
offset of the first float in the memory region to transfere
values the array to received the floats
start the first index in array to fill
count the maximum number of floats to copy

Throws
OffsetOutOfBoundsException when `offset` is negative or either `offset` or `offset` + `count` is greater than or equal to the size of this raw memory area.
ArrayIndexOutOfBoundsException when `start` is negative or either `start` or `start` + `count` is greater than or equal to the size of values.
NullPointerException when `values` is null or `count` is negative.

Returns
the number of floats actually transfered.

12.4.1.9 RawFloatWriter

A marker for a float accessor object encapsulating the protocol for writing floats from raw memory. A float accessor can always access at least one float. Each float is transfered in a single atomic operation. Groups of floats may be transfered together; however, this is not required.

Objects of this type are created with the method `RawMemoryFactory.createRawFloatWriter`[16] and `RawMemoryFactory.createRawFloat`[17]. Each object references a range of elements in the `RawMemoryRegion`[18] starting at the *base address* provided to the factory method. The size provided to the factor method determines the number of elements accessable.

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

Available since RTSJ 2.0

Interfaces
RawMemory

12.4.1.9.1 Methods

---

setFloat(float)

Set the value at the first position referenced by this instance, i.e., the value at its start address. This operation must be atomic with respect to all other raw memory accesses to the address. When an exception is thrown, no data is transfered.

Signature
```
public
void setFloat(float value)
```

setFloat(int, float)

Set the value of the n^{th} element referenced by this instance, where `n` is `offset` and the address is *base address* + `offset` ∗ *size of Float*. This operation must be atomic with respect to all other raw memory accesses to the address. When an exception is thrown, no data is transfered.

[16] Section 12.4.3.5.3
[17] Section 12.4.3.5.3
[18] Section 12.4.3.6

Signature

```
public
void setFloat(int offset, float value)
throws OffsetOutOfBoundsException
```

Parameters

offset of float in the memory region.

Throws

OffsetOutOfBoundsException when `offset` is negative or greater than or equal to the number of elements in the raw memory region.

set(int, float[])

Copy `values` to the raw memory starting at the address referenced by this instance plus the `offset` scaled by the element size in bytes and the objects `stride`. Only the floats in the intersection of `values` and the end of the memory region are transfered. When an exception is thrown, no data is transfered.

Signature

```
public
int set(int offset, float[] values)
throws OffsetOutOfBoundsException, NullPointerException
```

Throws

OffsetOutOfBoundsException when `offset` is negative or greater than or equal to the number of elements in the raw memory region.

NullPointerException when `values` is null.

Returns

the number of elements copied to `values`

set(int, float[], int, int)

Copy `values` to the memory region, where `offset` is first float in the memory region to write and `start` is the first index in `values` from which to read. The number of bytes transfered is the minimum of `count`, the *size* of the memory region minus `offset`, and length of `values` minus `start`. When an exception is thrown, no data is transfered.

Signature

```
public
int set(int offset, float[] values, int start, int count)
throws OffsetOutOfBoundsException,
ArrayIndexOutOfBoundsException, NullPointerException
```

Parameters

offset of the first float in the memory region to set
values the array from which to retrieve the floats
start the first index in array to fill
count the maximum number of floats to copy

Throws

OffsetOutOfBoundsException when `offset` is negative or either `offset` or `offset` + `count` is greater than or equal to the size of this raw memory area.
ArrayIndexOutOfBoundsException when `start` is negative or either `start` or `start` + `count` is greater than or equal to the size of values.
NullPointerException when `values` is null.

Returns

the number of floats actually transfered.

12.4.1.10 RawInt

A marker for an object that can be used to access to a single int. Read and write access to that int is checked by the factory that creates the instance; therefore, no access checking is provided by this interface, only bounds checking.
Available since RTSJ 2.0

Interfaces
RawIntReader
RawIntWriter

12.4.1.11 RawIntReader

A marker for a int accessor object encapsulating the protocol for reading ints from raw memory. A int accessor can always access at least one int. Each int is transfered in a single atomic operation. Groups of ints may be transfered together; however, this is not required.

Objects of this type are created with the method `RawMemoryFactory.createRawIntReader`[19] and `RawMemoryFactory.createRawInt`[20]. Each object references a range of elements in the `RawMemoryRegion`[21] starting at the *base address* provided to the factory method. The size provided to the factor method determines the number of elements accessable.

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

Available since RTSJ 2.0

Interfaces

RawMemory

12.4.1.11.1 Methods

---

getInt

Get the value at the first position referenced by this instance, i.e., the value at its start address. This operation must be atomic with respect to all other raw memory accesses to the address.

Signature

```
public
int getInt()
```

Returns

the value at the *base address* provided to the factory method that created this object.

getInt(int)

Get the value of the n^{th} element referenced by this instance, where `n` is `offset` and the address is *base address* + (`offset` ∗ stride ∗ *element size in bytes*). When an exception is thrown, no data is transfered.

[19]Section 12.4.3.5.3
[20]Section 12.4.3.5.3
[21]Section 12.4.3.6

Signature

```
public
int getInt(int offset)
throws OffsetOutOfBoundsException
```

Parameters

offset of int in the memory region starting from the address specified in the associated factory method.

Throws

OffsetOutOfBoundsException when `offset` is negative or greater than or equal to the number of elements in the raw memory region.

Returns

the value at the address specified.

get(int, int[])

Fill the `values` starting at the address referenced by this instance plus the offset scaled by the element size in bytes and the object's `stride`. Only the ints in the intersection of the start and end of `values` and the *base address* and the end of the memory region are transfered. When an exception is thrown, no data is transfered.

Signature

```
public
int get(int offset, int[] values)
throws OffsetOutOfBoundsException, NullPointerException
```

Parameters

offset of the first int in the memory region to transfere
values the array to received the ints

Throws

OffsetOutOfBoundsException when `offset` is negative or greater than or equal to the number of elements in the raw memory region.
NullPointerException when `values` is null.

Returns

the number of elements copied to `values`

get(int, int[], int, int)

Fill `values` with data from the memory region, where `offset` is first int in the memory region and `start` is the first index in `values`. The number of bytes transfered is the minimum of `count`, the *size* of the memory region minus `offset`, and length of `values` minus `start`. When an exception is thrown, no data is transfered.

Signature

```
public
int get(int offset, int[] values, int start, int count)
throws OffsetOutOfBoundsException,
ArrayIndexOutOfBoundsException, NullPointerException
```

Parameters

offset of the first int in the memory region to transfere
values the array to received the ints
start the first index in array to fill
count the maximum number of ints to copy

Throws

OffsetOutOfBoundsException when `offset` is negative or either `offset` or `offset` + `count` is greater than or equal to the size of this raw memory area.
ArrayIndexOutOfBoundsException when `start` is negative or either `start` or `start` + `count` is greater than or equal to the size of values.
NullPointerException when `values` is null or `count` is negative.

Returns

the number of ints actually transfered.

12.4.1.12 RawIntWriter

A marker for a int accessor object encapsulating the protocol for writing ints from raw memory. A int accessor can always access at least one int. Each int is transfered in a single atomic operation. Groups of ints may be transfered together; however, this is not required.

Objects of this type are created with the method `RawMemoryFactory.createRawIntWriter`[22] and `RawMemoryFactory.createRawInt`[23]. Each object references a range of elements in the `RawMemoryRegion`[24] starting at the *base address* provided to the factory

[22] Section 12.4.3.5.3
[23] Section 12.4.3.5.3
[24] Section 12.4.3.6

method. The size provided to the factor method determines the number of elements accessable.

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

Available since RTSJ 2.0

Interfaces
RawMemory

12.4.1.12.1 Methods

---

setInt(int)

Set the value at the first position referenced by this instance, i.e., the value at its start address. This operation must be atomic with respect to all other raw memory accesses to the address. When an exception is thrown, no data is transfered.

Signature

```
public
void setInt(int value)
```

setInt(int, int)

Set the value of the nth element referenced by this instance, where `n` is `offset` and the address is *base address* + `offset` ∗ *size of Int*. This operation must be atomic with respect to all other raw memory accesses to the address. When an exception is thrown, no data is transfered.

Signature

```
public
void setInt(int offset, int value)
throws OffsetOutOfBoundsException
```

Parameters
offset of int in the memory region.

Throws

OffsetOutOfBoundsException when `offset` is negative or greater than or equal to the number of elements in the raw memory region.

set(int, int[])

Copy `values` to the raw memory starting at the address referenced by this instance plus the `offset` scaled by the element size in bytes and the objects `stride`. Only the ints in the intersection of `values` and the end of the memory region are transfered. When an exception is thrown, no data is transfered.

Signature

```
public
int set(int offset, int[] values)
throws OffsetOutOfBoundsException, NullPointerException
```

Throws

OffsetOutOfBoundsException when `offset` is negative or greater than or equal to the number of elements in the raw memory region.
NullPointerException when `values` is null.

Returns

the number of elements copied to `values`

set(int, int[], int, int)

Copy `values` to the memory region, where `offset` is first int in the memory region to write and `start` is the first index in `values` from which to read. The number of bytes transfered is the minimum of `count`, the *size* of the memory region minus `offset`, and length of `values` minus `start`. When an exception is thrown, no data is transfered.

Signature

```
public
int set(int offset, int[] values, int start, int count)
throws OffsetOutOfBoundsException,
ArrayIndexOutOfBoundsException, NullPointerException
```

Parameters

offset of the first int in the memory region to set
values the array from which to retrieve the ints

start the first index in array to fill
count the maximum number of ints to copy

Throws

OffsetOutOfBoundsException when `offset` is negative or either `offset` or `offset` + `count` is greater than or equal to the size of this raw memory area.
ArrayIndexOutOfBoundsException when `start` is negative or either `start` or `start` + `count` is greater than or equal to the size of values.
NullPointerException when `values` is null.

Returns

the number of ints actually transfered.

12.4.1.13 RawLong

A marker for an object that can be used to access to a single long. Read and write access to that long is checked by the factory that creates the instance; therefore, no access checking is provided by this interface, only bounds checking.
Available since RTSJ 2.0

Interfaces
RawLongReader
RawLongWriter

12.4.1.14 RawLongReader

A marker for a long accessor object encapsulating the protocol for reading longs from raw memory. A long accessor can always access at least one long. Each long is transfered in a single atomic operation. Groups of longs may be transfered together; however, this is not required.

Objects of this type are created with the method `RawMemoryFactory.createRawLongReader`[25] and `RawMemoryFactory.createRawLong`[26]. Each object references a range of elements in the `RawMemoryRegion`[27] starting at the *base address* provided to the factory method. The size provided to the factor method determines the number of elements accessable.

[25] Section 12.4.3.5.3
[26] Section 12.4.3.5.3
[27] Section 12.4.3.6

Caching of the memory access is controlled by the factory that created this object. If the memory is not cached, this method guarantees serialized access. In other words, the memory access at the memory occurs in the same order as in the program. Multiple writes to the same location may not be coalesced.
Available since RTSJ 2.0

Interfaces
RawMemory

12.4.1.14.1 Methods

---

getLong

Get the value at the first position referenced by this instance, i.e., the value at its start address. This operation must be atomic with respect to all other raw memory accesses to the address.

Signature

```
public
long getLong()
```

Returns
the value at the *base address* provided to the factory method that created this object.

getLong(int)

Get the value of the n^{th} element referenced by this instance, where `n` is `offset` and the address is *base address* + (`offset` ∗ stride ∗ *element size in bytes*). When an exception is thrown, no data is transfered.

Signature

```
public
long getLong(int offset)
throws OffsetOutOfBoundsException
```

Parameters

offset of long in the memory region starting from the address specified in the associated factory method.

Throws

OffsetOutOfBoundsException when `offset` is negative or greater than or equal to the number of elements in the raw memory region.

Returns

the value at the address specified.

get(int, long[])

Fill the `values` starting at the address referenced by this instance plus the offset scaled by the element size in bytes and the object's `stride`. Only the longs in the intersection of the start and end of `values` and the *base address* and the end of the memory region are transfered. When an exception is thrown, no data is transfered.

Signature

```
public
int get(int offset, long[] values)
throws OffsetOutOfBoundsException, NullPointerException
```

Parameters

offset of the first long in the memory region to transfere

values the array to received the longs

Throws

OffsetOutOfBoundsException when `offset` is negative or greater than or equal to the number of elements in the raw memory region.

NullPointerException when `values` is null.

Returns

the number of elements copied to `values`

get(int, long[], int, int)

Fill `values` with data from the memory region, where `offset` is first long in the memory region and `start` is the first index in `values`. The number of bytes transfered is the minimum of `count`, the *size* of the memory region minus `offset`, and length of `values` minus `start`. When an exception is thrown, no data is transfered.

Signature

```
public
int get(int offset, long[] values, int start, int count)
throws OffsetOutOfBoundsException,
ArrayIndexOutOfBoundsException, NullPointerException
```

Parameters

offset of the first long in the memory region to transfere
values the array to received the longs
start the first index in array to fill
count the maximum number of longs to copy

Throws

OffsetOutOfBoundsException when `offset` is negative or either `offset` or `offset` + `count` is greater than or equal to the size of this raw memory area.
ArrayIndexOutOfBoundsException when `start` is negative or either `start` or `start` + `count` is greater than or equal to the size of values.
NullPointerException when `values` is null or `count` is negative.

Returns

the number of longs actually transfered.

12.4.1.15 RawLongWriter

A marker for a long accessor object encapsulating the protocol for writing longs from raw memory. A long accessor can always access at least one long. Each long is transfered in a single atomic operation. Groups of longs may be transfered together; however, this is not required.

Objects of this type are created with the method `RawMemoryFactory.createRawLongWriter`[28] and `RawMemoryFactory.createRawLong`[29]. Each object references a range of elements in the `RawMemoryRegion`[30] starting at the *base address* provided to the factory method. The size provided to the factor method determines the number of elements accessable.

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

Available since RTSJ 2.0

Interfaces

[28] Section 12.4.3.5.3
[29] Section 12.4.3.5.3
[30] Section 12.4.3.6

RawMemory

12.4.1.15.1 Methods

setLong(long)

Set the value at the first position referenced by this instance, i.e., the value at its start address. This operation must be atomic with respect to all other raw memory accesses to the address. When an exception is thrown, no data is transfered.

Signature

```
public
void setLong(long value)
```

setLong(int, long)

Set the value of the nth element referenced by this instance, where `n` is `offset` and the address is *base address* + `offset` ∗ *size of Long*. This operation must be atomic with respect to all other raw memory accesses to the address. When an exception is thrown, no data is transfered.

Signature

```
public
void setLong(int offset, long value)
throws OffsetOutOfBoundsException
```

Parameters

offset of long in the memory region.

Throws

OffsetOutOfBoundsException when `offset` is negative or greater than or equal to the number of elements in the raw memory region.

set(int, long[])

Copy `values` to the raw memory starting at the address referenced by this instance plus the `offset` scaled by the element size in bytes and the objects `stride`.

Only the longs in the intersection of `values` and the end of the memory region are transfered. When an exception is thrown, no data is transfered.

Signature

```
public
int set(int offset, long[] values)
throws OffsetOutOfBoundsException, NullPointerException
```

Throws

OffsetOutOfBoundsException when `offset` is negative or greater than or equal to the number of elements in the raw memory region.

NullPointerException when `values` is null.

Returns

the number of elements copied to `values`

set(int, long[], int, int)

Copy `values` to the memory region, where `offset` is first long in the memory region to write and `start` is the first index in `values` from which to read. The number of bytes transfered is the minimum of `count`, the *size* of the memory region minus `offset`, and length of `values` minus `start`. When an exception is thrown, no data is transfered.

Signature

```
public
int set(int offset, long[] values, int start, int count)
throws OffsetOutOfBoundsException,
ArrayIndexOutOfBoundsException, NullPointerException
```

Parameters

offset of the first long in the memory region to set

values the array from which to retrieve the longs

start the first index in array to fill

count the maximum number of longs to copy

Throws

OffsetOutOfBoundsException when `offset` is negative or either `offset` or `offset` + `count` is greater than or equal to the size of this raw memory area.

ArrayIndexOutOfBoundsException when `start` is negative or either `start` or `start` + `count` is greater than or equal to the size of values.

NullPointerException when `values` is null.

Returns

the number of longs actually transfered.

12.4.1.16 RawMemory

A marker for all raw memory accessor objects.
Available since RTSJ 2.0

12.4.1.16.1 Methods

getAddress

Get the base physical address of this object.

Signature

```
public
long getAddress()
```

Returns

the first physical address this raw memory object can access.

getSize

Get the number of bytes that this object spans.

Signature

```
public
int getSize()
```

Returns

the size of this raw memory

12.4.1.17 RawMemoryRegionFactory

A class to give an application the ability to provide support for a `RawMemoryRegion`[31] that is not already provided by the standard. An instance of this call can be registered with a `RawMemoryFactory`[32] and provides the object that that factory should return for a given
Available since RTSJ 2.0

12.4.1.17.1 Methods

---

getRegion

Get the region for which this factory creates raw memory objects.

Signature

```
public
javax.realtime.device.RawMemoryRegion getRegion()
```

getName

Get the name of the region for which this factory creates raw memory objects.

Signature

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

createRawByte(long, int, int)

Create an instance of a class that implements `RawByte`[33] and accesses memory of `getRegion`[34] in the address range described by `base`, `stride`, and `count`. The actual extent of the memory addressed by the object is `stride` ∗ *size of RawByte* ∗ `count`. The object is allocated in the current memory area of the calling thread.

[31] Section 12.4.3.6
[32] Section 12.4.3.5
[33] Section 12.4.1.1
[34] Section 12.4.1.17.1

Available since RTSJ 2.0

Signature

```
public
javax.realtime.device.RawByte createRawByte(long base, int
count, int stride)
throws SecurityException, OffsetOutOfBoundsException,
SizeOutOfBoundsException, UnsupportedRawMemoryRegionException,
MemoryTypeConflictException
```

Parameters

base The starting physical address accessible through the returned instance.
count The number of memory elements accessible through the returned instance.
stride The distance to the next element in mulitple of element count, where a value of `1` means the elements are adjacent in memory.

Throws

IllegalArgumentException when `base` is negative, `count` is not greater than zero, or `stride` is not greater than zero.
SecurityException when the caller does not have permissions to access the given memory region or the specified range of addresses.
OffsetOutOfBoundsException when `base` is invalid.
SizeOutOfBoundsException when the memory addressed by the object would extend into an invalid range of memory.
MemoryTypeConflictException when `base` does not point to memory that matches the type served by this factory.

Returns

an object that implements `RawByte`[35] and supports access to the specified range in the memory region.

createRawByteReader(long, int, int)

Create an instance of a class that implements `RawByteReader`[36] and accesses memory of `getRegion`[37] in the address range described by `base`, `stride`, and `count`. The actual extent of the memory addressed by the object is `stride` ∗ *size of RawByteReader* ∗ `count`. The object is allocated in the current memory area of the

[35] Section 12.4.1.1
[36] Section 12.4.1.2
[37] Section 12.4.1.17.1

calling thread.
Available since RTSJ 2.0

Signature

```
public
javax.realtime.device.RawByteReader createRawByteReader(long
base, int count, int stride)
throws SecurityException, OffsetOutOfBoundsException,
SizeOutOfBoundsException, UnsupportedRawMemoryRegionException,
MemoryTypeConflictException
```

Parameters

base The starting physical address accessible through the returned instance.
count The number of memory elements accessible through the returned instance.
stride The distance to the next element in mulitple of element count, where a value of `1` means the elements are adjacent in memory.

Throws

IllegalArgumentException when `base` is negative, `count` is not greater than zero, or `stride` is not greater than zero.
SecurityException when the caller does not have permissions to access the given memory region or the specified range of addresses.
OffsetOutOfBoundsException when `base` is invalid.
SizeOutOfBoundsException when the memory addressed by the object would extend into an invalid range of memory.
MemoryTypeConflictException when `base` does not point to memory that matches the type served by this factory.

Returns

an object that implements `RawByteReader`[38] and supports access to the specified range in the memory region.

createRawByteWriter(long, int, int)

Create an instance of a class that implements `RawByteWriter`[39] and accesses memory of `getRegion`[40] in the address range described by `base`, `stride`, and `count`.

[38]Section 12.4.1.2
[39]Section 12.4.1.3
[40]Section 12.4.1.17.1

The actual extent of the memory addressed by the object is `stride` ∗ *size of RawByteWriter* ∗ `count`. The object is allocated in the current memory area of the calling thread.

Available since RTSJ 2.0

Signature

```
public
javax.realtime.device.RawByteWriter createRawByteWriter(long
base, int count, int stride)
throws SecurityException, OffsetOutOfBoundsException,
SizeOutOfBoundsException, UnsupportedRawMemoryRegionException,
MemoryTypeConflictException
```

Parameters

base The starting physical address accessible through the returned instance.

count The number of memory elements accessible through the returned instance.

stride The distance to the next element in mulitple of element count, where a value of `1` means the elements are adjacent in memory.

Throws

IllegalArgumentException when `base` is negative, `count` is not greater than zero, or `stride` is not greater than zero.

SecurityException when the caller does not have permissions to access the given memory region or the specified range of addresses.

OffsetOutOfBoundsException when `base` is invalid.

SizeOutOfBoundsException when the memory addressed by the object would extend into an invalid range of memory.

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

Returns

an object that implements `RawByteWriter`[41] and supports access to the specified range in the memory region.

createRawShort(long, int, int)

Create an instance of a class that implements `RawShort`[42] and accesses memory

[41]Section 12.4.1.3
[42]Section 12.4.1.18

of `getRegion`[43] in the address range described by `base`, `stride`, and `count`. The actual extent of the memory addressed by the object is `stride` ∗ *size of RawShort* ∗ `count`. The object is allocated in the current memory area of the calling thread.
Available since RTSJ 2.0

Signature

`public`
`javax.realtime.device.RawShort createRawShort(long base, int count, int stride)`
`throws SecurityException, OffsetOutOfBoundsException, SizeOutOfBoundsException, UnsupportedRawMemoryRegionException, MemoryTypeConflictException`

Parameters

base The starting physical address accessible through the returned instance.
count The number of memory elements accessible through the returned instance.
stride The distance to the next element in mulitple of element count, where a value of `1` means the elements are adjacent in memory.

Throws

IllegalArgumentException when `base` is negative, `count` is not greater than zero, or `stride` is not greater than zero.
SecurityException when the caller does not have permissions to access the given memory region or the specified range of addresses.
OffsetOutOfBoundsException when `base` is invalid.
SizeOutOfBoundsException when the memory addressed by the object would extend into an invalid range of memory.
MemoryTypeConflictException when `base` does not point to memory that matches the type served by this factory.

Returns

an object that implements `RawShort`[44] and supports access to the specified range in the memory region.

createRawShortReader(long, int, int)

Create an instance of a class that implements `RawShortReader`[45] and accesses

[43] Section 12.4.1.17.1
[44] Section 12.4.1.18
[45] Section 12.4.1.19

memory of `getRegion`[46] in the address range described by `base`, `stride`, and `count`. The actual extent of the memory addressed by the object is `stride` ∗ *size of RawShortReader* ∗ `count`. The object is allocated in the current memory area of the calling thread.

Available since RTSJ 2.0

Signature

```
public
javax.realtime.device.RawShortReader createRawShortReader(long
base, int count, int stride)
throws SecurityException, OffsetOutOfBoundsException,
SizeOutOfBoundsException, UnsupportedRawMemoryRegionException,
MemoryTypeConflictException
```

Parameters

base The starting physical address accessible through the returned instance.
count The number of memory elements accessible through the returned instance.
stride The distance to the next element in mulitple of element count, where a value of `1` means the elements are adjacent in memory.

Throws

IllegalArgumentException when `base` is negative, `count` is not greater than zero, or `stride` is not greater than zero.
SecurityException when the caller does not have permissions to access the given memory region or the specified range of addresses.
OffsetOutOfBoundsException when `base` is invalid.
SizeOutOfBoundsException when the memory addressed by the object would extend into an invalid range of memory.
MemoryTypeConflictException when `base` does not point to memory that matches the type served by this factory.

Returns

an object that implements `RawShortReader`[47] and supports access to the specified range in the memory region.

createRawShortWriter(long, int, int)

[46] Section 12.4.1.17.1
[47] Section 12.4.1.19

Create an instance of a class that implements `RawShortWriter`[48] and accesses memory of `getRegion`[49] in the address range described by `base`, `stride`, and `count`. The actual extent of the memory addressed by the object is `stride` ∗ *size of RawShortWriter* ∗ `count`. The object is allocated in the current memory area of the calling thread.

Available since RTSJ 2.0

Signature

```
public
javax.realtime.device.RawShortWriter createRawShortWriter(long
base, int count, int stride)
throws SecurityException, OffsetOutOfBoundsException,
SizeOutOfBoundsException, UnsupportedRawMemoryRegionException,
MemoryTypeConflictException
```

Parameters

base The starting physical address accessible through the returned instance.
count The number of memory elements accessible through the returned instance.
stride The distance to the next element in mulitple of element count, where a value of `1` means the elements are adjacent in memory.

Throws

IllegalArgumentException when `base` is negative, `count` is not greater than zero, or `stride` is not greater than zero.
SecurityException when the caller does not have permissions to access the given memory region or the specified range of addresses.
OffsetOutOfBoundsException when `base` is invalid.
SizeOutOfBoundsException when the memory addressed by the object would extend into an invalid range of memory.
MemoryTypeConflictException when `base` does not point to memory that matches the type served by this factory.

Returns

an object that implements `RawShortWriter`[50] and supports access to the specified range in the memory region.

createRawInt(long, int, int)

[48] Section 12.4.1.20
[49] Section 12.4.1.17.1
[50] Section 12.4.1.20

Create an instance of a class that implements `RawInt`[51] and accesses memory of `getRegion`[52] in the address range described by `base`, `stride`, and `count`. The actual extent of the memory addressed by the object is `stride` ∗ *size of RawInt* ∗ `count`. The object is allocated in the current memory area of the calling thread.
Available since RTSJ 2.0

Signature

```
public
javax.realtime.device.RawInt createRawInt(long base, int count,
int stride)
throws SecurityException, OffsetOutOfBoundsException,
SizeOutOfBoundsException, UnsupportedRawMemoryRegionException,
MemoryTypeConflictException
```

Parameters

base The starting physical address accessible through the returned instance.
count The number of memory elements accessible through the returned instance.
stride The distance to the next element in mulitple of element count, where a value of `1` means the elements are adjacent in memory.

Throws

IllegalArgumentException when `base` is negative, `count` is not greater than zero, or `stride` is not greater than zero.
SecurityException when the caller does not have permissions to access the given memory region or the specified range of addresses.
OffsetOutOfBoundsException when `base` is invalid.
SizeOutOfBoundsException when the memory addressed by the object would extend into an invalid range of memory.
MemoryTypeConflictException when `base` does not point to memory that matches the type served by this factory.

Returns

an object that implements `RawInt`[53] and supports access to the specified range in the memory region.

createRawIntReader(long, int, int)

[51]Section 12.4.1.10
[52]Section 12.4.1.17.1
[53]Section 12.4.1.10

Create an instance of a class that implements `RawIntReader`[54] and accesses memory of `getRegion`[55] in the address range described by `base`, `stride`, and `count`. The actual extent of the memory addressed by the object is `stride` ∗ *size of RawIntReader* ∗ `count`. The object is allocated in the current memory area of the calling thread.

Available since RTSJ 2.0

Signature

```
public
javax.realtime.device.RawIntReader createRawIntReader(long base,
int count, int stride)
throws SecurityException, OffsetOutOfBoundsException,
SizeOutOfBoundsException, UnsupportedRawMemoryRegionException,
MemoryTypeConflictException
```

Parameters

base The starting physical address accessible through the returned instance.
count The number of memory elements accessible through the returned instance.
stride The distance to the next element in mulitple of element count, where a value of `1` means the elements are adjacent in memory.

Throws

IllegalArgumentException when `base` is negative, `count` is not greater than zero, or `stride` is not greater than zero.
SecurityException when the caller does not have permissions to access the given memory region or the specified range of addresses.
OffsetOutOfBoundsException when `base` is invalid.
SizeOutOfBoundsException when the memory addressed by the object would extend into an invalid range of memory.
MemoryTypeConflictException when `base` does not point to memory that matches the type served by this factory.

Returns

an object that implements `RawIntReader`[56] and supports access to the specified range in the memory region.

[54] Section 12.4.1.11
[55] Section 12.4.1.17.1
[56] Section 12.4.1.11

createRawIntWriter(long, int, int)

Create an instance of a class that implements `RawIntWriter`[57] and accesses memory of `getRegion`[58] in the address range described by `base`, `stride`, and `count`. The actual extent of the memory addressed by the object is `stride` ∗ *size of RawIntWriter* ∗ `count`. The object is allocated in the current memory area of the calling thread.

Available since RTSJ 2.0

Signature

```
public
javax.realtime.device.RawIntWriter createRawIntWriter(long base,
int count, int stride)
throws SecurityException, OffsetOutOfBoundsException,
SizeOutOfBoundsException, UnsupportedRawMemoryRegionException,
MemoryTypeConflictException
```

Parameters

base The starting physical address accessible through the returned instance.

count The number of memory elements accessible through the returned instance.

stride The distance to the next element in mulitple of element count, where a value of `1` means the elements are adjacent in memory.

Throws

IllegalArgumentException when `base` is negative, `count` is not greater than zero, or `stride` is not greater than zero.

SecurityException when the caller does not have permissions to access the given memory region or the specified range of addresses.

OffsetOutOfBoundsException when `base` is invalid.

SizeOutOfBoundsException when the memory addressed by the object would extend into an invalid range of memory.

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

Returns

an object that implements `RawIntWriter`[59] and supports access to the specified range in the memory region.

[57] Section 12.4.1.12
[58] Section 12.4.1.17.1
[59] Section 12.4.1.12

createRawLong(long, int, int)

Create an instance of a class that implements `RawLong`[60] and accesses memory of `getRegion`[61] in the address range described by `base`, `stride`, and `count`. The actual extent of the memory addressed by the object is `stride` $*$ *size of RawLong* $*$ `count`. The object is allocated in the current memory area of the calling thread.
Available since RTSJ 2.0

Signature

```
public
javax.realtime.device.RawLong createRawLong(long base, int
count, int stride)
throws SecurityException, OffsetOutOfBoundsException,
SizeOutOfBoundsException, UnsupportedRawMemoryRegionException,
MemoryTypeConflictException
```

Parameters

base The starting physical address accessible through the returned instance.
count The number of memory elements accessible through the returned instance.
stride The distance to the next element in mulitple of element count, where a value of `1` means the elements are adjacent in memory.

Throws

IllegalArgumentException when `base` is negative, `count` is not greater than zero, or `stride` is not greater than zero.
SecurityException when the caller does not have permissions to access the given memory region or the specified range of addresses.
OffsetOutOfBoundsException when `base` is invalid.
SizeOutOfBoundsException when the memory addressed by the object would extend into an invalid range of memory.
MemoryTypeConflictException when `base` does not point to memory that matches the type served by this factory.

Returns

an object that implements `RawLong`[62] and supports access to the specified range in the memory region.

[60]Section 12.4.1.13
[61]Section 12.4.1.17.1
[62]Section 12.4.1.13

createRawLongReader(long, int, int)

Create an instance of a class that implements `RawLongReader`[63] and accesses memory of `getRegion`[64] in the address range described by `base`, `stride`, and `count`. The actual extent of the memory addressed by the object is `stride` ∗ *size of RawLongReader* ∗ `count`. The object is allocated in the current memory area of the calling thread.

Available since RTSJ 2.0

Signature

```
public
javax.realtime.device.RawLongReader createRawLongReader(long
base, int count, int stride)
throws SecurityException, OffsetOutOfBoundsException,
SizeOutOfBoundsException, UnsupportedRawMemoryRegionException,
MemoryTypeConflictException
```

Parameters

base The starting physical address accessible through the returned instance.

count The number of memory elements accessible through the returned instance.

stride The distance to the next element in mulitple of element count, where a value of `1` means the elements are adjacent in memory.

Throws

IllegalArgumentException when `base` is negative, `count` is not greater than zero, or `stride` is not greater than zero.

SecurityException when the caller does not have permissions to access the given memory region or the specified range of addresses.

OffsetOutOfBoundsException when `base` is invalid.

SizeOutOfBoundsException when the memory addressed by the object would extend into an invalid range of memory.

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

Returns

an object that implements `RawLongReader`[65] and supports access to the specified range in the memory region.

[63] Section 12.4.1.14
[64] Section 12.4.1.17.1
[65] Section 12.4.1.14

createRawLongWriter(long, int, int)

Create an instance of a class that implements `RawLongWriter`[66] and accesses memory of `getRegion`[67] in the address range described by `base`, `stride`, and `count`. The actual extent of the memory addressed by the object is `stride` ∗ *size of RawLongWriter* ∗ `count`. The object is allocated in the current memory area of the calling thread.
Available since RTSJ 2.0

Signature

```
public
javax.realtime.device.RawLongWriter createRawLongWriter(long
base, int count, int stride)
throws SecurityException, OffsetOutOfBoundsException,
SizeOutOfBoundsException, UnsupportedRawMemoryRegionException,
MemoryTypeConflictException
```

Parameters

base The starting physical address accessible through the returned instance.
count The number of memory elements accessible through the returned instance.
stride The distance to the next element in mulitple of element count, where a value of `1` means the elements are adjacent in memory.

Throws

IllegalArgumentException when `base` is negative, `count` is not greater than zero, or `stride` is not greater than zero.
SecurityException when the caller does not have permissions to access the given memory region or the specified range of addresses.
OffsetOutOfBoundsException when `base` is invalid.
SizeOutOfBoundsException when the memory addressed by the object would extend into an invalid range of memory.
MemoryTypeConflictException when `base` does not point to memory that matches the type served by this factory.

Returns

an object that implements `RawLongWriter`[68] and supports access to the specified range in the memory region.

[66]Section 12.4.1.15
[67]Section 12.4.1.17.1
[68]Section 12.4.1.15

createRawFloat(long, int, int)

Create an instance of a class that implements RawFloat[69] and accesses memory of getRegion[70] in the address range described by base, stride, and count. The actual extent of the memory addressed by the object is stride ∗ *size of RawFloat* ∗ count. The object is allocated in the current memory area of the calling thread.
Available since RTSJ 2.0

Signature

```
public
javax.realtime.device.RawFloat createRawFloat(long base, int
count, int stride)
throws SecurityException, OffsetOutOfBoundsException,
SizeOutOfBoundsException, UnsupportedRawMemoryRegionException,
MemoryTypeConflictException
```

Parameters

base The starting physical address accessible through the returned instance.
count The number of memory elements accessible through the returned instance.
stride The distance to the next element in mulitple of element count, where a value of 1 means the elements are adjacent in memory.

Throws

IllegalArgumentException when base is negative, count is not greater than zero, or stride is not greater than zero.
SecurityException when the caller does not have permissions to access the given memory region or the specified range of addresses.
OffsetOutOfBoundsException when base is invalid.
SizeOutOfBoundsException when the memory addressed by the object would extend into an invalid range of memory.
MemoryTypeConflictException when base does not point to memory that matches the type served by this factory.

Returns

an object that implements RawFloat[71] and supports access to the specified range in the memory region.

[69] Section 12.4.1.7
[70] Section 12.4.1.17.1
[71] Section 12.4.1.7

createRawFloatReader(long, int, int)

Create an instance of a class that implements `RawFloatReader`[72] and accesses memory of `getRegion`[73] in the address range described by `base`, `stride`, and `count`. The actual extent of the memory addressed by the object is `stride` ∗ *size of RawFloatReader* ∗ `count`. The object is allocated in the current memory area of the calling thread.

Available since RTSJ 2.0

Signature

```
public
javax.realtime.device.RawFloatReader createRawFloatReader(long
base, int count, int stride)
throws SecurityException, OffsetOutOfBoundsException,
SizeOutOfBoundsException, UnsupportedRawMemoryRegionException,
MemoryTypeConflictException
```

Parameters

base The starting physical address accessible through the returned instance.

count The number of memory elements accessible through the returned instance.

stride The distance to the next element in mulitple of element count, where a value of `1` means the elements are adjacent in memory.

Throws

IllegalArgumentException when `base` is negative, `count` is not greater than zero, or `stride` is not greater than zero.

SecurityException when the caller does not have permissions to access the given memory region or the specified range of addresses.

OffsetOutOfBoundsException when `base` is invalid.

SizeOutOfBoundsException when the memory addressed by the object would extend into an invalid range of memory.

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

Returns

an object that implements `RawFloatReader`[74] and supports access to the specified range in the memory region.

[72]Section 12.4.1.8
[73]Section 12.4.1.17.1
[74]Section 12.4.1.8

createRawFloatWriter(long, int, int)

Create an instance of a class that implements `RawFloatWriter`[75] and accesses memory of `getRegion`[76] in the address range described by `base`, `stride`, and `count`. The actual extent of the memory addressed by the object is `stride` ∗ *size of RawFloatWriter* ∗ `count`. The object is allocated in the current memory area of the calling thread.

Available since RTSJ 2.0

Signature

```
public
javax.realtime.device.RawFloatWriter createRawFloatWriter(long
base, int count, int stride)
throws SecurityException, OffsetOutOfBoundsException,
SizeOutOfBoundsException, UnsupportedRawMemoryRegionException,
MemoryTypeConflictException
```

Parameters

base The starting physical address accessible through the returned instance.

count The number of memory elements accessible through the returned instance.

stride The distance to the next element in mulitple of element count, where a value of `1` means the elements are adjacent in memory.

Throws

IllegalArgumentException when `base` is negative, `count` is not greater than zero, or `stride` is not greater than zero.

SecurityException when the caller does not have permissions to access the given memory region or the specified range of addresses.

OffsetOutOfBoundsException when `base` is invalid.

SizeOutOfBoundsException when the memory addressed by the object would extend into an invalid range of memory.

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

Returns

an object that implements `RawFloatWriter`[77] and supports access to the specified range in the memory region.

[75] Section 12.4.1.9
[76] Section 12.4.1.17.1
[77] Section 12.4.1.9

createRawDouble(long, int, int)

Create an instance of a class that implements `RawDouble`[78] and accesses memory of `getRegion`[79] in the address range described by `base`, `stride`, and `count`. The actual extent of the memory addressed by the object is `stride` ∗ *size of RawDouble* ∗ `count`. The object is allocated in the current memory area of the calling thread.
Available since RTSJ 2.0

Signature

```
public
javax.realtime.device.RawDouble createRawDouble(long base, int
count, int stride)
throws SecurityException, OffsetOutOfBoundsException,
SizeOutOfBoundsException, UnsupportedRawMemoryRegionException,
MemoryTypeConflictException
```

Parameters

base The starting physical address accessible through the returned instance.
count The number of memory elements accessible through the returned instance.
stride The distance to the next element in mulitple of element count, where a value of `1` means the elements are adjacent in memory.

Throws

IllegalArgumentException when `base` is negative, `count` is not greater than zero, or `stride` is not greater than zero.
SecurityException when the caller does not have permissions to access the given memory region or the specified range of addresses.
OffsetOutOfBoundsException when `base` is invalid.
SizeOutOfBoundsException when the memory addressed by the object would extend into an invalid range of memory.
MemoryTypeConflictException when `base` does not point to memory that matches the type served by this factory.

Returns

an object that implements `RawDouble`[80] and supports access to the specified range in the memory region.

[78] Section 12.4.1.4
[79] Section 12.4.1.17.1
[80] Section 12.4.1.4

createRawDoubleReader(long, int, int)

Create an instance of a class that implements `RawDoubleReader`[81] and accesses memory of `getRegion`[82] in the address range described by `base`, `stride`, and `count`. The actual extent of the memory addressed by the object is `stride` ∗ *size of RawDoubleReader* ∗ `count`. The object is allocated in the current memory area of the calling thread.
Available since RTSJ 2.0

Signature

```
public
javax.realtime.device.RawDoubleReader createRawDoubleReader(long
base, int count, int stride)
throws SecurityException, OffsetOutOfBoundsException,
SizeOutOfBoundsException, UnsupportedRawMemoryRegionException,
MemoryTypeConflictException
```

Parameters

base The starting physical address accessible through the returned instance.
count The number of memory elements accessible through the returned instance.
stride The distance to the next element in mulitple of element count, where a value of `1` means the elements are adjacent in memory.

Throws

IllegalArgumentException when `base` is negative, `count` is not greater than zero, or `stride` is not greater than zero.
SecurityException when the caller does not have permissions to access the given memory region or the specified range of addresses.
OffsetOutOfBoundsException when `base` is invalid.
SizeOutOfBoundsException when the memory addressed by the object would extend into an invalid range of memory.
MemoryTypeConflictException when `base` does not point to memory that matches the type served by this factory.

Returns

an object that implements `RawDoubleReader`[83] and supports access to the specified range in the memory region.

[81] Section 12.4.1.5
[82] Section 12.4.1.17.1
[83] Section 12.4.1.5

createRawDoubleWriter(long, int, int)

Create an instance of a class that implements `RawDoubleWriter`[84] and accesses memory of `getRegion`[85] in the address range described by `base`, `stride`, and `count`. The actual extent of the memory addressed by the object is `stride` ∗ *size of RawDoubleWriter* ∗ `count`. The object is allocated in the current memory area of the calling thread.

Available since RTSJ 2.0

Signature

```
public
javax.realtime.device.RawDoubleWriter createRawDoubleWriter(long
base, int count, int stride)
throws SecurityException, OffsetOutOfBoundsException,
SizeOutOfBoundsException, UnsupportedRawMemoryRegionException,
MemoryTypeConflictException
```

Parameters

base The starting physical address accessible through the returned instance.

count The number of memory elements accessible through the returned instance.

stride The distance to the next element in mulitple of element count, where a value of `1` means the elements are adjacent in memory.

Throws

IllegalArgumentException when `base` is negative, `count` is not greater than zero, or `stride` is not greater than zero.

SecurityException when the caller does not have permissions to access the given memory region or the specified range of addresses.

OffsetOutOfBoundsException when `base` is invalid.

SizeOutOfBoundsException when the memory addressed by the object would extend into an invalid range of memory.

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

Returns

an object that implements `RawDoubleWriter`[86] and supports access to the specified range in the memory region.

[84] Section 12.4.1.6
[85] Section 12.4.1.17.1
[86] Section 12.4.1.6

12.4.1.18 RawShort

A marker for an object that can be used to access to a single short. Read and write access to that short is checked by the factory that creates the instance; therefore, no access checking is provided by this interface, only bounds checking.
Available since RTSJ 2.0

Interfaces
RawShortReader
RawShortWriter

12.4.1.19 RawShortReader

A marker for a short accessor object encapsulating the protocol for reading shorts from raw memory. A short accessor can always access at least one short. Each short is transfered in a single atomic operation. Groups of shorts may be transfered together; however, this is not required.

Objects of this type are created with the method `RawMemoryFactory.createRawShortReader`[87] and `RawMemoryFactory.createRawShort`[88]. Each object references a range of elements in the `RawMemoryRegion`[89] starting at the *base address* provided to the factory method. The size provided to the factor method determines the number of elements accessable.

Caching of the memory access is controlled by the factory that created this object. If the memory is not cached, this method guarantees serialized access. In other words, the memory access at the memory occurs in the same order as in the program. Multiple writes to the same location may not be coalesced.
Available since RTSJ 2.0

Interfaces
RawMemory

12.4.1.19.1 Methods

[87]Section 12.4.3.5.3
[88]Section 12.4.3.5.3
[89]Section 12.4.3.6

getShort

Get the value at the first position referenced by this instance, i.e., the value at its start address. This operation must be atomic with respect to all other raw memory accesses to the address.

Signature

```
public
short getShort()
```

Returns

the value at the *base address* provided to the factory method that created this object.

getShort(int)

Get the value of the nth element referenced by this instance, where `n` is `offset` and the address is *base address* + (`offset` ∗ stride ∗ *element size in bytes*). When an exception is thrown, no data is transfered.

Signature

```
public
short getShort(int offset)
throws OffsetOutOfBoundsException
```

Parameters

offset of short in the memory region starting from the address specified in the associated factory method.

Throws

OffsetOutOfBoundsException when `offset` is negative or greater than or equal to the number of elements in the raw memory region.

Returns

the value at the address specified.

get(int, short[])

Fill the `values` starting at the address referenced by this instance plus the offset scaled by the element size in bytes and the object's `stride`. Only the shorts in the intersection of the start and end of `values` and the *base address* and the end of the memory region are transfered. When an exception is thrown, no data is transfered.

Signature

```
public
int get(int offset, short[] values)
throws OffsetOutOfBoundsException, NullPointerException
```

Parameters

offset of the first short in the memory region to transfere
values the array to received the shorts

Throws

OffsetOutOfBoundsException when `offset` is negative or greater than or equal to the number of elements in the raw memory region.
NullPointerException when `values` is null.

Returns

the number of elements copied to `values`

get(int, short[], int, int)

Fill `values` with data from the memory region, where `offset` is first short in the memory region and `start` is the first index in `values`. The number of bytes transfered is the minimum of `count`, the *size* of the memory region minus `offset`, and length of `values` minus `start`. When an exception is thrown, no data is transfered.

Signature

```
public
int get(int offset, short[] values, int start, int count)
throws OffsetOutOfBoundsException,
ArrayIndexOutOfBoundsException, NullPointerException
```

Parameters

offset of the first short in the memory region to transfere
values the array to received the shorts
start the first index in array to fill
count the maximum number of shorts to copy

Throws

OffsetOutOfBoundsException when `offset` is negative or either `offset` or `offset` + `count` is greater than or equal to the size of this raw memory area.
ArrayIndexOutOfBoundsException when `start` is negative or either `start` or `start` + `count` is greater than or equal to the size of values.
NullPointerException when `values` is null or `count` is negative.

Returns

the number of shorts actually transfered.

12.4.1.20 RawShortWriter

A marker for a short accessor object encapsulating the protocol for writing shorts from raw memory. A short accessor can always access at least one short. Each short is transfered in a single atomic operation. Groups of shorts may be transfered together; however, this is not required.

Objects of this type are created with the method `RawMemoryFactory.createRawShortWriter`[90] and `RawMemoryFactory.createRawShort`[91]. Each object references a range of elements in the `RawMemoryRegion`[92] starting at the *base address* provided to the factory method. The size provided to the factor method determines the number of elements accessable.

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

Available since RTSJ 2.0

Interfaces

RawMemory

12.4.1.20.1 Methods

setShort(short)

Set the value at the first position referenced by this instance, i.e., the value at its start address. This operation must be atomic with respect to all other raw memory accesses to the address. When an exception is thrown, no data is transfered.

Signature

[90] Section 12.4.3.5.3
[91] Section 12.4.3.5.3
[92] Section 12.4.3.6

```
public
void setShort(short value)
```

setShort(int, short)

Set the value of the nth element referenced by this instance, where `n` is `offset` and the address is *base address* + `offset` ∗ *size of Short*. This operation must be atomic with respect to all other raw memory accesses to the address. When an exception is thrown, no data is transfered.

Signature

```
public
void setShort(int offset, short value)
throws OffsetOutOfBoundsException
```

Parameters

offset of short in the memory region.

Throws

OffsetOutOfBoundsException when `offset` is negative or greater than or equal to the number of elements in the raw memory region.

set(int, short[])

Copy `values` to the raw memory starting at the address referenced by this instance plus the `offset` scaled by the element size in bytes and the objects `stride`. Only the shorts in the intersection of `values` and the end of the memory region are transfered. When an exception is thrown, no data is transfered.

Signature

```
public
int set(int offset, short[] values)
throws OffsetOutOfBoundsException, NullPointerException
```

Throws

OffsetOutOfBoundsException when `offset` is negative or greater than or equal to the number of elements in the raw memory region.

NullPointerException when `values` is null.

Returns

the number of elements copied to `values`

set(int, short[], int, int)

Copy `values` to the memory region, where `offset` is first short in the memory region to write and `start` is the first index in `values` from which to read. The number of bytes transfered is the minimum of `count`, the *size* of the memory region minus `offset`, and length of `values` minus `start`. When an exception is thrown, no data is transfered.

Signature

```
public
int set(int offset, short[] values, int start, int count)
throws OffsetOutOfBoundsException,
ArrayIndexOutOfBoundsException, NullPointerException
```

Parameters

offset of the first short in the memory region to set
values the array from which to retrieve the shorts
start the first index in array to fill
count the maximum number of shorts to copy

Throws

OffsetOutOfBoundsException when `offset` is negative or either `offset` or `offset` + `count` is greater than or equal to the size of this raw memory area.
ArrayIndexOutOfBoundsException when `start` is negative or either `start` or `start` + `count` is greater than or equal to the size of values.
NullPointerException when `values` is null.

Returns

the number of shorts actually transfered.

12.4.2 Exceptions

12.4.2.1 UnsupportedRawMemoryRegionException

Inheritance

java.lang.Object
  java.lang.Throwable
    java.lang.Exception
      java.lang.RuntimeException
        javax.realtime.device.UnsupportedRawMemoryRegionException

12.4.2.1.1 Constructors

UnsupportedRawMemoryRegionException

Signature

```
public
  UnsupportedRawMemoryRegionException()
```

UnsupportedRawMemoryRegionException(String)

Signature

```
public
  UnsupportedRawMemoryRegionException(String s)
```

UnsupportedRawMemoryRegionException(Throwable)

Signature

```
public
  UnsupportedRawMemoryRegionException(Throwable ex)
```

UnsupportedRawMemoryRegionException(String, Throwable)

Signature

```
public
  UnsupportedRawMemoryRegionException(String s, Throwable ex)
```

12.4.3 Classes

12.4.3.1 Happening

This class provides second level handling for external events such as interrupts. A happening can be triggered by an `InterruptServiceRoutine`[93] or from native code. Application defined `Happenings` can be identified by an application provided name or a system provided `id`, both of which must be unique. A system `Happening` has a name provide by the system which is a string beginning with `@`.
Available since RTSJ 2.0

Inheritance

java.lang.Object
  javax.realtime.AbstractAsyncEvent
    javax.realtime.AsyncEvent
      javax.realtime.device.Happening

Interfaces

ActiveEvent

12.4.3.1.1 Constructors

Happening(String)

Create a Happening with the given name.

Signature

```
public
  Happening(String name)
```

[93] Section 12.4.3.3

```
throws IllegalArgumentException
```

Parameters

name of the happening.

Throws

IllegalArgumentException when name does not match the pattern full type naming convention, i.e., package plus name. An implementation may throw this exception for all names starting with java. and javax.

Happening(String, HappeningDispatcher)

Create a Happening with the given name.

Signature

```
public
  Happening(String name, HappeningDispatcher dispatcher)


throws IllegalArgumentException
```

Parameters

name of the happening.

dispatcher to use when being triggered.

Throws

IllegalArgumentException when name does not match the pattern full identifier naming convention, i.e., package plus name. An implementation may throw this exception for all names starting with java. and javax.

12.4.3.1.2 Methods

---

getHappening(String)

Find an active happening by its name. A `Happening` is only registered when it is active. When it is stopped, it is automatically deregistered.

Signature

```
public static
javax.realtime.device.Happening getHappening(String name)
```

Parameters

name of the happening to get.

Returns

a reference to the happening with name `name`, or `null` if no happening is found.

createID(String)

Sets up a mapping between a `name` and a system dependent ID. This can be called either in the constructor of an instance of `InterruptServiceRoutine`[94] or in native code that sets up an interrupt service routine to link it with a `Happening`. Once created, it cannot be removed.

This must take no more than linear time in the number of ID (`n`) registered, but should be `O(log2(n))`.

Signature

```
public static
int createID(String name)
throws IllegalStateException
```

Parameters

name is a happing name string.

Throws

IllegalStateException when `name` is already registered.

Returns

an ID assigned by the system

getID(String)

Return the ID of `name`, when one exists or -1, when `name` is not registered.

This must take no more than linear time in the number of ID (`n`) registered, but should be `O(log2(n))`.

Signature

```
public static
int getID(String name)
```

Parameters

name is a happening name string.

Returns

[94] Section 12.4.3.3

The ID, or -1 if no happening is found by that name.

isHappening(String)

Is there an active happening with name `name`?

Signature

```
public static
boolean isHappening(String name)
```

Parameters

name A string that might name an active happening.

Returns

True only when there is a registered happening with the name `name`.

trigger(int)

Causes the event dispatcher corresponding to `happeningId` to be scheduled for execution. The implementation should be simple enough so that it can be done in the context of an `InterrruptServiceRoutine.handle`[95] method.

`trigger()` and any native code analog to it interact with other `ActiveEvent`[96] code effectively as if `trigger()` signals a POSIX counting semaphore that the happening is waiting on.

The implementation is encouraged to create (and document) a native code analog to this method that can be used without a Java context.

This method must execute in constant time.

Signature

```
public static
boolean trigger(int id)
```

Parameters

id identifies which happening to trigger.

Returns

true if a happening with id `happeningId` was found, false otherwise.

[95] Section **??**

[96] Section 8.4.1.1

start

Start this `Happening`, i.e., change to the active and enabled state. Once a happening is started the first time, when it is in a scoped memory it increments the scope count of that scope; otherwise, it becomes a member of the root set. An active and enabled happening dispatches its handlers when fired.
See Section stop()

Signature

```
public
void start()
throws IllegalStateException
```

Throws

IllegalStateException when this `Happening` has already been started or its `name` is already in use by another happening that has been started.

start(boolean)

Start this `Happening`, but leave it in the disabled state. When fired before being enabled, it does not dispatch its handlers.
See Section stop()

Signature

```
public
void start(boolean disabled)
throws IllegalStateException
```

Parameters

disabled true for starting in a disabled state.

Throws

IllegalStateException when this `Happening` has already been started.

stop

Stop this `happening` from responding to the `fire` and `trigger` methods.

Signature

```
public
boolean stop()
throws IllegalStateException
```

Throws

IllegalStateException when this `Happening` is not active.

Returns

`true` when `this` is in the *enabled* state `false` otherwise.

getId

Get the number of this happening.

Signature

```
public
int getId()
```

Returns

the happening number or -1, when not registered.

getName

Get the name of this happening.

Signature

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

Returns

the name of this happening.

isActive

Determine the activation state of this happening, i.e., it has been started.

Signature

```
public
boolean isActive()
```

Returns

`true` when active, `false` otherwise.

isRunning

Determine whether or not this `Happening` is both active an enabled.

Signature

```
public
boolean isRunning()
```

Returns

`true` when this `Happening` is both active and enables, `false` otherwise.

12.4.3.2 HappeningDispatcher

This class provides a means of dispatching a set of `Happening`[97]. The process() method is called each time the signal is triggered.

Inheritance

java.lang.Object
  javax.realtime.ActiveEventDispatcher
    javax.realtime.device.HappeningDispatcher

12.4.3.2.1 Constructors

HappeningDispatcher(SchedulingParameters)

Create a new dispatcher, whose dispatching thread runs with the given scheduling parameters.

Signature

```
public
  HappeningDispatcher(SchedulingParameters schedule)
```

[97] Section 12.4.3.1

Parameters

schedule give the parameters for scheduling this dispatcher

12.4.3.2.2 Methods

register(Happening)

Register a Happening[98] with this dispatcher.

Signature

```
public synchronized
void register(Happening happening)
throws RegistrationException, IllegalStateException,
IllegalArgumentException
```

Parameters

happening to register

Throws

RegistrationException when `happening` is already registered.
IllegalStateException when this object has been destroyed.
IllegalArgumentException when `happening` is not stopped.

deregister(Happening)

Deregister a Happening[99] form this dispatcher.

Signature

```
public synchronized
void deregister(Happening happening)
throws DeregistrationException, IllegalStateException,
IllegalArgumentException
```

Parameters

happening to unregister

Throws

DeregistrationException when `happening` is not already registered.

[98] Section 12.4.3.1
[99] Section 12.4.3.1

IllegalStateException when this object has been destroyed.
IllegalArgumentException when `happening` is not stopped.

destroy

Release all reasources thereby making the dispatcher unusable.

Signature

```
public
void destroy()
throws IllegalStateException
```

Throws

IllegalStateException when called on a dispatcher that has one or more registered `Happening`[100] objects.

12.4.3.3 InterruptServiceRoutine

A class for defining a first level interrupt handler. The implementation must override the `handle`[101] method to provide the code to be run when an interrupt occurs. This class must always be present in the Device module, but may do nothing in a context that does not provide direct access to interrupts, e.g., in user space on an operating system that does not support user space device drivers.

Inheritance

java.lang.Object
  javax.realtime.device.InterruptServiceRoutine

12.4.3.3.1 Constructors

InterruptServiceRoutine(MemoryArea)

Create an interrupt service routine with a particular memory area.

[100] Section 12.4.3.1
[101] Section 12.4.3.3.2

Signature

```
public
  InterruptServiceRoutine(MemoryArea area)


 throws NullPointerException, IllegalArgumentException
```

Parameters

area the allocation context in which the `handle`[102] method runs.

Throws

NullPointerException when `initial_area` is null.

IllegalArgumentException when `id` is not a valid interrupt id.

12.4.3.3.2 Methods

---

getMaximumInterruptPriority

Retrieve the maximum interrupt priority. It must be greater than or equal to the result of `getMinimumInterruptPriority`[103].

Signature

```
public static
int getMaximumInterruptPriority()
```

Returns

the maximum interrupt priority.

getMinimumInterruptPriority

Retrieve the minimum interrupt priority. It must be higher than all other priorities provided by the system.

Signature

```
public static
int getMinimumInterruptPriority()
```

[102]Section 12.4.3.3.2
[103]Section 12.4.3.3.2

Returns

the minimum interrupt priority.

getHandler(int)

Find the `InterruptServiceRoutine` that is handling a given interrupt.

Signature

```
public static
javax.realtime.device.InterruptServiceRoutine getHandler(int
interrupt)
```

Parameters

interrupt for which to find the `InterruptServiceRoutine`

Returns

the `InterruptServiceRoutine` registered to the given interrupt. `Null` is returned when nothing is registered for that interrupt.

getInterruptPriority(int)

Get the interrupt priority of a given interrupt.

Signature

```
public static
int getInterruptPriority(int interruptId)
throws IllegalArgumentException
```

Throws

IllegalArgumentException when there is no interrupt corresponding to `interruptId`

Returns

the priority at which the `handle`[104] method is involked. The returned value is always greater than `PriorityScheduler.getMaxPriority()`[105].

register(int)

Register this interrupt service routine with the system so that it can be triggered.

[104] Section 12.4.3.3.2
[105] Section 6.4.2.6.3

Signature

```
public
void register(int interrupt)
throws RegistrationException
```

Parameters

interrupt a system dependent identifier for the interrupt.

Throws

RegistrationException when `this` is already registered or some other `InterruptServiceRoutine`[106] is registered for `interrupt`.

deregister

Deregister this interrupt service routine with the system so that it can no longer be triggered.

Signature

```
public
void deregister()
throws DeregistrationException
```

Throws

DeregistrationException when this interrupt service routine is not registered.

isRegistered

A predicate for the registration state.

Signature

```
public final
boolean isRegistered()
```

Returns

true when registered, otherwise false.

handle

[106] Section 12.4.3.3

The code to execute for first level interrupt handling. A subclass defines this to give the required behavior. `RawMemory`[107] classes may be used to access the associated device registers and a `Happening`[108] may be triggered for second level interrupt handling.

The code used to implement this method should not block itself for an unbound amount of time or induce a context switch, e.g., sleeping. The effects of unbound blocking and inducing a context switch here are undefined and could result in deadlocking the machine. `Object.notify()` and `Object.notifyAll()` may be called, but `Object.wait()` should not be called.

Unless the overridden method is synchronized, the infrastructure shall provide no synchronization for the execution of this method. When the `MemoryArea`[109] provided at creation is a `ScopedMemory`[110], its count is incremented on entry to this method and decremented on exit.

Signature

```
protected abstract
void handle()
```

12.4.3.4 RawBufferFactory

A factory class for generating raw byte buffers. This enables the infrastructure to limit the address ranges from which a buffer may be taken.

Inheritance

java.lang.Object
  javax.realtime.device.RawBufferFactory

12.4.3.4.1 Constructors

RawBufferFactory(long, long)

Create a factory for allocating buffers in a particular address range. Whether the address is physical or virtual is system dependent.

[107] Section 12.4.1.16
[108] Section 12.4.3.1
[109] Section 11.4.3.6
[110] Section 11.4.3.11

Open issue: jjh—I am not sure how to handle the issue of mapping addresses that the system knows about to ones that can be used in a DMA controller or other driver that requires buffers. Alternatives include providing more information to the factory or having a translation function to get the "right" address. An implementation may need or need to provide a means of mapping a physical page into virtual memory. **End of open issue**

Signature

```
public
  RawBufferFactory(long base, long size)


 throws MemoryInUseException
```

Parameters

base is the base address of a memory range for buffer allocation

size is the number of bytes in the memory range

Throws

MemoryInUseException when the memory area provide is already in use by or reserved for a `MemoryArea`[111], program code, or other sytem or VM structure.

12.4.3.4.2 Methods

---

allocateDirectByteBuffer(int)

Create a direct byte buffer with the given capacity within the range of this factory.

Open issue: jjh—what if a subclass is needed? One could add a reflection call, but it would be hard to implement in general. **End of open issue**

Signature

```
public
java.nio.ByteBuffer allocateDirectByteBuffer(int capacity)
```

Parameters

capacity the number of bytes in the buffer.

[111]Section 11.4.3.6

Returns
the new buffer.

defineDirectByteBuffer(long, int)

Given a range of memory within the allocation area defined by this factory, create a direct byte buffer to represent that memory range.

Signature
```
public
java.nio.ByteBuffer defineDirectByteBuffer(long start, int
capacity)
throws RangeOutOfBoundsException
```
Parameters
start is the beginning of the memory range
capacity is number of bytes in the range
Throws
RangeOutOfBoundsException when `start` or `start + capacity` extends outside of the allocation area of this factory.
Returns
the new buffer object

inRange(ByteBuffer)

Check to see if the buffer's data area is within the range of this factory.

Signature
```
public
boolean inRange(ByteBuffer buffer)
```
Parameters
buffer to check
Returns
`true` when and only when buffer's data area is within the range of this factory; otherwise `false`

addressOf(ByteBuffer)

Give the location of this buffers data in memory. The address shall be in the address space of the DMA controller.

Signature

```
public
long addressOf(ByteBuffer buffer)
```

Parameters

buffer

Returns

the start address of the data range of this buffer

writeFence(ByteBuffer)

Ensures that all changes to the `DirectByteBuffer` buffer by the current thread have been flushed in a manner that makes them visible to other threads (including native threads), and behaves as a volatile store with respect to the Java Memory Model synchronization order.

This method should invoke a memory barrier operation that is understood by the VM, runtime, native compiler, and platform to provide visibility to all changes to the associated buffer made before its invocation.

Signature

```
public static
void writeFence(ByteBuffer buffer)
```

Parameters

buffer the byte buffer which will be flushed

readFence(ByteBuffer)

Ensures that any previous changes to the memory represented by the given `DirectByteBuffer` by other threads (including native threads) will be visible when it is next accessed by the current thread, and behaves as a volatile load with respect to the Java Memory Model synchronization order.

This method should invoke a memory barrier operation that is understood by the VM, runtime, native compiler, and platform to provide visibility for any changes to the associated buffer previously flushed with a call to `writeFence(ByteBuffer buffer)`[112] or its native equivalent on the buffer's memory.

[112]Section 12.4.3.4.2

Signature

```
public static
void readFence(ByteBuffer buffer)
```

Parameters

buffer the byte buffer which will be updated

12.4.3.5 RawMemoryFactory

This class is the hub of a system that constructs special purpose objects to access particular types and ranges of raw memory. This facility is supported by the `register(RawMemoryRegionFactory)`[113] methods. An application developer can use this method to add support for additional memory regions.

Each create method returns an object of the corresponding type, e.g., the `createRawByte(RawMemoryRegion, long, int, int)`[114] method returns a reference to an object that implements the `RawByte`[115] interface and supports access to the requested type of memory and address range. Each create method is permitted to optimize error checking and access based on the requested memory type and address range.

The useage pattern for raw memory, assuming the necessary factory has been registered, is illustrated by this example.

```
// Get an accessor object that can access memory starting at
// baseAddress, for size bytes.
RawInt memory =
  RawMemoryFactory.createRawInt(RawMemoryFactory.MEMORY_MAPPED_REGION,
                                address, count, stride, false);
// Use the accessor to load from and store to raw memory.
int loadedData = memory.getInt(someOffset);
memory.setInt(otherOffset, intVal);
```

When an application needs to access a class of memory that is not already supported by a registered factory, the developer must define a memory region by implementing a factory which can create objects to access memory in that region. Thus, the application must implement a factory that

[113] Section 12.4.3.5.3
[114] Section 12.4.3.5.3
[115] Section 12.4.1.1

implements the `RawMemoryRegionFactory`[116] interface.

A raw memory region factory is identified by a `RawMemoryRegion`[117] that is used by each create method, e.g., `createRawByte(RawMemoryRegion, long, int, int)`[118], to locate the appropriate factory. The name is not passed to `register(RawMemoryRegionFactory)` as a separate argument, rather the name is provided to `register(RawMemoryRegionFactory)`[120] through the factory's `RawMemoryRegionFactory.getName`[121] method.

The `register(RawMemoryRegionFactory)`[122] method is only used when by application code when it needs to add support for a new type of raw memory.

Whether a give `offset` addresses a high-order or low-order byte of an aligned `short` in memory is determined by the value of the `RealtimeSystem.BYTE_ORDER`[123] static byte variable in class `RealtimeSystem`[124], the start address of the object, the `offset` given the `stride` of the object. Regardless of the byte ordering, accessor methods for by continue to select bytes starting at `offset` from the base address and continuing toward greater addresses.

The `RawMemory` class enables a realtime program to implement device drivers, memory-mapped I/O, flash memory, battery-backed RAM, and similar low-level software.

A raw memory region cannot contain references to Java objects. Such a capability would be unsafe (since it could be used to defeat Java's type checking) and error prone (since it is sensitive to the specific representational choices made by the Java compiler).

Atomic loads and stores on raw memory are defined in terms of physical memory. This memory may be accessible to threads outside the JVM and to non-programmed access (e.g., DMA). Consequently, atomic access must be supported by hardware. This specification is written with the assumption that all suitable hardware platforms support atomic loads from raw memory for aligned bytes, shorts, and ints. Atomic access beyond the specified minimum may be supported by the implementation.

Storing values into raw memory is more hardware-dependent than loading values. Many processor architectures do not support atomic stores of variables except for aligned stores of the processor's word size. For instance, storing a byte into memory might require reading a 32-bit quantity into a processor register, updating

[116] Section 12.4.1.17
[117] Section 12.4.3.6
[118] Section 12.4.3.5.3
[119] Section 12.4.3.5.3
[120] Section 12.4.3.5.3
[121] Section 12.4.1.17.1
[122] Section 12.4.3.5.3
[123] Section 13.3.1.7.1
[124] Section 13.3.1.7

the register to reflect the new byte value, then restoring the whole 32-bit quantity. Changes to other bytes in the 32-bit quantity that take place between the load and the store are lost.

Some processors have mechanisms that can be used to implement an atomic store of a byte, but those mechanisms are often slow and not universally supported.

This class need not support unaligned access to data; but if it does, it is not require the implementation to make such access atomic. Accesses to data aligned on its natural boundary will be atomic if the processor implements atomic loads and stores of that data size.

Except where noted, accesses to raw memory are not atomic with respect to the memory or with respect to schedulable objects. A raw memory region could be updated by another schedulable object, or even unmapped in the middle of an access method, or even *removed* mid method.

The characteristics of raw-memory access are necessarily platform dependent. This specification provides a minimum requirement for the RTSJ platform, but it also supports optional system properties that identify a platform's level of support for atomic raw put and get. The properties represent a four-dimensional sparse array of access type, data type, alignment, and atomicity with boolean values indicating whether that combination of access attributes is atomic. The default value for array entries is false. The dimension are

Attribute	Values	Comment
Access type	read, write	
Data type	• byte, • short, • int, • long, • float, • double	
Alignment	0 to 7	For each data type, the possible alignments range from • 0 == aligned • to data size - 1 == only the first byte of the data is *alignment* bytes away from natural alignment.
Atomicity	• processor, • smp, • memory	• *processor* means access is atomic with respect to other schedulable objects on that processor. • *smp* means that access is *processor* atomic, and atomic across the processors in an SMP. • *memory* means that access is *smp* atomic, and atomic with respect to all access to the memory, including DMA hardware.

The true values in the table are represented by properties of the following form. javax.realtime.atomicaccess_<access>_<type>_<alignment>_atomicity=true for example,

javax.realtime.atomicaccess_read_byte_0_memory=true

Table entries with a value of false may be explicitly represented, but since false is the default value, such properties are redundant.

All raw memory access is treated as volatile, and *serialized*. The runtime must be

forced to read memory or write to memory on each call to a raw memory objects's getter or setter method, and to complete the reads and writes in the order they appear in the program order.
Available since RTSJ 2.0

Inheritance

java.lang.Object
javax.realtime.device.RawMemoryFactory

12.4.3.5.1 Fields

MEMORY_MAPPED_REGION

`public static final MEMORY_MAPPED_REGION`

This raw memory name is predefined for use to request access to memory mapped I/O devices.

IO_PORT_MAPPED_REGION

`public static final IO_PORT_MAPPED_REGION`

This raw memory name is predefined for use to request access to I/O device space implemented by processor instructions, such as the x86 `in` and `out` instructions.

12.4.3.5.2 Constructors

RawMemoryFactory

Create an empty factory. For a factory with support for the platform defined `RawMemoryRegion`[125]s, use `getDefaultFactory`[126] instead.

Signature

[125]Section 12.4.3.6
[126]Section 12.4.3.5.3

```
public
  RawMemoryFactory()
```

12.4.3.5.3 Methods

getDefaultFactory

Get the factory with support for the platform defined regions.

Signature

```
public static
javax.realtime.device.RawMemoryFactory getDefaultFactory()
```

register(RawMemoryRegionFactory)

Add support for a new memory region

Signature

```
public
void register(RawMemoryRegionFactory factory)
throws RegistrationException
```

Parameters

factory is the `RawMemoryRegionFactory`[127] to use for creating `RawMemory`[128] objects for the the `RawMemoryRegion`[129] it makes available.

Throws

RegistrationException when the `RawMemoryRegion` of `factory` already has a factory registered.

deregister(RawMemoryRegionFactory)

Remove support for a new memory region

[127] Section 12.4.1.17
[128] Section 12.4.1.16
[129] Section 12.4.3.6

Signature

```
public static
void deregister(RawMemoryRegionFactory factory)
throws DeregistrationException
```

Parameters

factory is the `RawMemoryRegionFactory`[130] to make unavailable.

Throws

RegistrationException when the `factory` is not registered.

createRawByte(RawMemoryRegion, long, int, int)

Create an instance of a class that implements `RawByte`[131] and accesses memory of `region` in the address range described by `base`, `stride`, and `count`. The actual extent of the memory addressed by the object is `stride` $*$ *size of RawByte* $*$ `count`. The object is allocated in the current memory area of the calling thread.

Signature

```
public
javax.realtime.device.RawByte createRawByte(RawMemoryRegion
region, long base, int count, int stride)
throws SecurityException, OffsetOutOfBoundsException,
SizeOutOfBoundsException, MemoryTypeConflictException,
UnsupportedRawMemoryRegionException
```

Parameters

base The starting physical address accessible through the returned instance.

count The number of memory elements accessible through the returned instance.

stride The distance to the next element as a mulitple of element size, where `1` means the elements are adjacent in memory.

Throws

IllegalArgumentException when `base` is negative, `count` is not greater than zero, or `stride` is less than one.

SecurityException when the caller does not have permissions to access the given memory region or the specified range of addresses.

OffsetOutOfBoundsException when `base` is invalid.

[130] Section 12.4.1.17

[131] Section 12.4.1.1

SizeOutOfBoundsException when the memory addressed by the object would extend into an invalid range of memory.
MemoryTypeConflictException when `base` does not point to memory that matches the type served by this factory.

Returns
an object that implements `RawByte`[132] and supports access to the specified range in the memory region.

createRawByteReader(RawMemoryRegion, long, int, int)

Create an instance of a class that implements `RawByteReader`[133] and accesses memory of `region` in the address range described by `base`, `stride`, and `count`. The actual extent of the memory addressed by the object is `stride` ∗ *size of RawByteReader* ∗ `count`. The object is allocated in the current memory area of the calling thread.

Signature
```
public
javax.realtime.device.RawByteReader createRawByte-
Reader(RawMemoryRegion region, long base, int count, int stride)
throws SecurityException, OffsetOutOfBoundsException,
SizeOutOfBoundsException, MemoryTypeConflictException,
UnsupportedRawMemoryRegionException
```

Parameters
base The starting physical address accessible through the returned instance.
count The number of memory elements accessible through the returned instance.
stride The distance to the next element as a mulitple of element size, where `1` means the elements are adjacent in memory.

Throws
IllegalArgumentException when `base` is negative, `count` is not greater than zero, or `stride` is less than one.
SecurityException when the caller does not have permissions to access the given memory region or the specified range of addresses.
OffsetOutOfBoundsException when `base` is invalid.
SizeOutOfBoundsException when the memory addressed by the object would extend into an invalid range of memory.

[132] Section 12.4.1.1
[133] Section 12.4.1.2

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

Returns

an object that implements `RawByteReader`[134] and supports access to the specified range in the memory region.

createRawByteWriter(RawMemoryRegion, long, int, int)

Create an instance of a class that implements `RawByteWriter`[135] and accesses memory of `region` in the address range described by `base`, `stride`, and `count`. The actual extent of the memory addressed by the object is `stride` ∗ *size of RawByteWriter* ∗ `count`. The object is allocated in the current memory area of the calling thread.

Signature

```
public
javax.realtime.device.RawByteWriter createRawByteWri-
ter(RawMemoryRegion region, long base, int count, int stride)
throws SecurityException, OffsetOutOfBoundsException,
SizeOutOfBoundsException, MemoryTypeConflictException,
UnsupportedRawMemoryRegionException
```

Parameters

base The starting physical address accessible through the returned instance.

count The number of memory elements accessible through the returned instance.

stride The distance to the next element as a mulitple of element size, where `1` means the elements are adjacent in memory.

Throws

IllegalArgumentException when `base` is negative, `count` is not greater than zero, or `stride` is less than one.

SecurityException when the caller does not have permissions to access the given memory region or the specified range of addresses.

OffsetOutOfBoundsException when `base` is invalid.

SizeOutOfBoundsException when the memory addressed by the object would extend into an invalid range of memory.

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

[134] Section 12.4.1.2

[135] Section 12.4.1.3

Returns

an object that implements `RawByteWriter`[136] and supports access to the specified range in the memory region.

createRawShort(RawMemoryRegion, long, int, int)

Create an instance of a class that implements `RawShort`[137] and accesses memory of `region` in the address range described by `base`, `stride`, and `count`. The actual extent of the memory addressed by the object is `stride` ∗ *size of RawShort* ∗ `count`. The object is allocated in the current memory area of the calling thread.

Signature

```
public
javax.realtime.device.RawShort createRawShort(RawMemoryRegion
region, long base, int count, int stride)
throws SecurityException, OffsetOutOfBoundsException,
SizeOutOfBoundsException, MemoryTypeConflictException,
UnsupportedRawMemoryRegionException
```

Parameters

base The starting physical address accessible through the returned instance.
count The number of memory elements accessible through the returned instance.
stride The distance to the next element as a mulitple of element size, where `1` means the elements are adjacent in memory.

Throws

IllegalArgumentException when `base` is negative, `count` is not greater than zero, or `stride` is less than one.
SecurityException when the caller does not have permissions to access the given memory region or the specified range of addresses.
OffsetOutOfBoundsException when `base` is invalid.
SizeOutOfBoundsException when the memory addressed by the object would extend into an invalid range of memory.
MemoryTypeConflictException when `base` does not point to memory that matches the type served by this factory.

Returns

[136] Section 12.4.1.3
[137] Section 12.4.1.18

an object that implements `RawShort`[138] and supports access to the specified range in the memory region.

createRawShortReader(RawMemoryRegion, long, int, int)

Create an instance of a class that implements `RawShortReader`[139] and accesses memory of `region` in the address range described by `base`, `stride`, and `count`. The actual extent of the memory addressed by the object is `stride` ∗ *size of RawShortReader* ∗ `count`. The object is allocated in the current memory area of the calling thread.

Signature

```
public
javax.realtime.device.RawShortReader createRawShort-
Reader(RawMemoryRegion region, long base, int count, int stride)
throws SecurityException, OffsetOutOfBoundsException,
SizeOutOfBoundsException, MemoryTypeConflictException,
UnsupportedRawMemoryRegionException
```

Parameters

base The starting physical address accessible through the returned instance.
count The number of memory elements accessible through the returned instance.
stride The distance to the next element as a mulitple of element size, where `1` means the elements are adjacent in memory.

Throws

IllegalArgumentException when `base` is negative, `count` is not greater than zero, or `stride` is less than one.
SecurityException when the caller does not have permissions to access the given memory region or the specified range of addresses.
OffsetOutOfBoundsException when `base` is invalid.
SizeOutOfBoundsException when the memory addressed by the object would extend into an invalid range of memory.
MemoryTypeConflictException when `base` does not point to memory that matches the type served by this factory.

Returns

an object that implements `RawShortReader`[140] and supports access to the

[138] Section 12.4.1.18
[139] Section 12.4.1.19
[140] Section 12.4.1.19

specified range in the memory region.

createRawShortWriter(RawMemoryRegion, long, int, int)

Create an instance of a class that implements `RawShortWriter`[141] and accesses memory of `region` in the address range described by `base`, `stride`, and `count`. The actual extent of the memory addressed by the object is `stride` ∗ *size of RawShortWriter* ∗ `count`. The object is allocated in the current memory area of the calling thread.

Signature

```
public
javax.realtime.device.RawShortWriter createRawShortWri-
ter(RawMemoryRegion region, long base, int count, int stride)
throws SecurityException, OffsetOutOfBoundsException,
SizeOutOfBoundsException, MemoryTypeConflictException,
UnsupportedRawMemoryRegionException
```

Parameters

base The starting physical address accessible through the returned instance.

count The number of memory elements accessible through the returned instance.

stride The distance to the next element as a mulitple of element size, where `1` means the elements are adjacent in memory.

Throws

IllegalArgumentException when `base` is negative, `count` is not greater than zero, or `stride` is less than one.

SecurityException when the caller does not have permissions to access the given memory region or the specified range of addresses.

OffsetOutOfBoundsException when `base` is invalid.

SizeOutOfBoundsException when the memory addressed by the object would extend into an invalid range of memory.

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

Returns

an object that implements `RawShortWriter`[142] and supports access to the specified range in the memory region.

[141] Section 12.4.1.20

[142] Section 12.4.1.20

createRawInt(RawMemoryRegion, long, int, int)

Create an instance of a class that implements `RawInt`[143] and accesses memory of `region` in the address range described by `base`, `stride`, and `count`. The actual extent of the memory addressed by the object is `stride` ∗ *size of RawInt* ∗ `count`. The object is allocated in the current memory area of the calling thread.

Signature

```
public
javax.realtime.device.RawInt createRawInt(RawMemoryRegion
region, long base, int count, int stride)
throws SecurityException, OffsetOutOfBoundsException,
SizeOutOfBoundsException, MemoryTypeConflictException,
UnsupportedRawMemoryRegionException
```

Parameters

base The starting physical address accessible through the returned instance.
count The number of memory elements accessible through the returned instance.
stride The distance to the next element as a mulitple of element size, where `1` means the elements are adjacent in memory.

Throws

IllegalArgumentException when `base` is negative, `count` is not greater than zero, or `stride` is less than one.
SecurityException when the caller does not have permissions to access the given memory region or the specified range of addresses.
OffsetOutOfBoundsException when `base` is invalid.
SizeOutOfBoundsException when the memory addressed by the object would extend into an invalid range of memory.
MemoryTypeConflictException when `base` does not point to memory that matches the type served by this factory.

Returns

an object that implements `RawInt`[144] and supports access to the specified range in the memory region.

createRawIntReader(RawMemoryRegion, long, int, int)

[143] Section 12.4.1.10
[144] Section 12.4.1.10

Create an instance of a class that implements `RawIntReader`[145] and accesses memory of `region` in the address range described by `base`, `stride`, and `count`. The actual extent of the memory addressed by the object is `stride` ∗ *size of RawIntReader* ∗ `count`. The object is allocated in the current memory area of the calling thread.

Signature

```
public
javax.realtime.device.RawIntReader createRawInt-
Reader(RawMemoryRegion region, long base, int count, int stride)
throws SecurityException, OffsetOutOfBoundsException,
SizeOutOfBoundsException, MemoryTypeConflictException,
UnsupportedRawMemoryRegionException
```

Parameters

base The starting physical address accessible through the returned instance.
count The number of memory elements accessible through the returned instance.
stride The distance to the next element as a mulitple of element size, where `1` means the elements are adjacent in memory.

Throws

IllegalArgumentException when `base` is negative, `count` is not greater than zero, or `stride` is less than one.
SecurityException when the caller does not have permissions to access the given memory region or the specified range of addresses.
OffsetOutOfBoundsException when `base` is invalid.
SizeOutOfBoundsException when the memory addressed by the object would extend into an invalid range of memory.
MemoryTypeConflictException when `base` does not point to memory that matches the type served by this factory.

Returns

an object that implements `RawIntReader`[146] and supports access to the specified range in the memory region.

createRawIntWriter(RawMemoryRegion, long, int, int)

Create an instance of a class that implements `RawIntWriter`[147] and accesses

[145] Section 12.4.1.11
[146] Section 12.4.1.11
[147] Section 12.4.1.12

memory of `region` in the address range described by `base`, `stride`, and `count`. The actual extent of the memory addressed by the object is `stride` ∗ *size of RawIntWriter* ∗ `count`. The object is allocated in the current memory area of the calling thread.

Signature

```
public
javax.realtime.device.RawIntWriter createRawIntWri-
ter(RawMemoryRegion region, long base, int count, int stride)
throws SecurityException, OffsetOutOfBoundsException,
SizeOutOfBoundsException, MemoryTypeConflictException,
UnsupportedRawMemoryRegionException
```

Parameters

base The starting physical address accessible through the returned instance.
count The number of memory elements accessible through the returned instance.
stride The distance to the next element as a mulitple of element size, where `1` means the elements are adjacent in memory.

Throws

IllegalArgumentException when `base` is negative, `count` is not greater than zero, or `stride` is less than one.
SecurityException when the caller does not have permissions to access the given memory region or the specified range of addresses.
OffsetOutOfBoundsException when `base` is invalid.
SizeOutOfBoundsException when the memory addressed by the object would extend into an invalid range of memory.
MemoryTypeConflictException when `base` does not point to memory that matches the type served by this factory.

Returns

an object that implements `RawIntWriter`[148] and supports access to the specified range in the memory region.

createRawLong(RawMemoryRegion, long, int, int)

Create an instance of a class that implements `RawLong`[149] and accesses memory of `region` in the address range described by `base`, `stride`, and `count`. The actual

[148] Section 12.4.1.12
[149] Section 12.4.1.13

extent of the memory addressed by the object is `stride` ∗ *size of RawLong* ∗ `count`. The object is allocated in the current memory area of the calling thread.

Signature

```
public
javax.realtime.device.RawLong createRawLong(RawMemoryRegion
region, long base, int count, int stride)
throws SecurityException, OffsetOutOfBoundsException,
SizeOutOfBoundsException, MemoryTypeConflictException,
UnsupportedRawMemoryRegionException
```

Parameters

base The starting physical address accessible through the returned instance.
count The number of memory elements accessible through the returned instance.
stride The distance to the next element as a mulitple of element size, where `1` means the elements are adjacent in memory.

Throws

IllegalArgumentException when `base` is negative, `count` is not greater than zero, or `stride` is less than one.
SecurityException when the caller does not have permissions to access the given memory region or the specified range of addresses.
OffsetOutOfBoundsException when `base` is invalid.
SizeOutOfBoundsException when the memory addressed by the object would extend into an invalid range of memory.
MemoryTypeConflictException when `base` does not point to memory that matches the type served by this factory.

Returns

an object that implements `RawLong`[150] and supports access to the specified range in the memory region.

createRawLongReader(RawMemoryRegion, long, int, int)

Create an instance of a class that implements `RawLongReader`[151] and accesses memory of `region` in the address range described by `base`, `stride`, and `count`. The actual extent of the memory addressed by the object is `stride` ∗ *size of RawLongReader* ∗ `count`. The object is allocated in the current memory area of the calling thread.

[150] Section 12.4.1.13
[151] Section 12.4.1.14

Signature

`public`
`javax.realtime.device.RawLongReader`
`createRawLongReader(RawMemoryRegion region, long base, int count, int stride)`
`throws SecurityException, OffsetOutOfBoundsException, SizeOutOfBoundsException, MemoryTypeConflictException, UnsupportedRawMemoryRegionException`

Parameters

base The starting physical address accessible through the returned instance.
count The number of memory elements accessible through the returned instance.
stride The distance to the next element as a mulitple of element size, where `1` means the elements are adjacent in memory.

Throws

IllegalArgumentException when `base` is negative, `count` is not greater than zero, or `stride` is less than one.
SecurityException when the caller does not have permissions to access the given memory region or the specified range of addresses.
OffsetOutOfBoundsException when `base` is invalid.
SizeOutOfBoundsException when the memory addressed by the object would extend into an invalid range of memory.
MemoryTypeConflictException when `base` does not point to memory that matches the type served by this factory.

Returns

an object that implements `RawLongReader`[152] and supports access to the specified range in the memory region.

createRawLongWriter(RawMemoryRegion, long, int, int)

Create an instance of a class that implements `RawLongWriter`[153] and accesses memory of `region` in the address range described by `base`, `stride`, and `count`. The actual extent of the memory addressed by the object is `stride` ∗ *size of RawLongWriter* ∗ `count`. The object is allocated in the current memory area of the calling thread.

[152] Section 12.4.1.14
[153] Section 12.4.1.15

Signature

```
public
javax.realtime.device.RawLongWriter createRawLong-
Writer(RawMemoryRegion region, long base, int count, int stride)
throws SecurityException, OffsetOutOfBoundsException,
SizeOutOfBoundsException, MemoryTypeConflictException,
UnsupportedRawMemoryRegionException
```

Parameters

base The starting physical address accessible through the returned instance.
count The number of memory elements accessible through the returned instance.
stride The distance to the next element as a mulitple of element size, where `1` means the elements are adjacent in memory.

Throws

IllegalArgumentException when `base` is negative, `count` is not greater than zero, or `stride` is less than one.
SecurityException when the caller does not have permissions to access the given memory region or the specified range of addresses.
OffsetOutOfBoundsException when `base` is invalid.
SizeOutOfBoundsException when the memory addressed by the object would extend into an invalid range of memory.
MemoryTypeConflictException when `base` does not point to memory that matches the type served by this factory.

Returns

an object that implements `RawLongWriter`[154] and supports access to the specified range in the memory region.

createRawFloat(RawMemoryRegion, long, int, int)

Create an instance of a class that implements `RawFloat`[155] and accesses memory of `region` in the address range described by `base`, `stride`, and `count`. The actual extent of the memory addressed by the object is `stride` ∗ *size of RawFloat* ∗ `count`. The object is allocated in the current memory area of the calling thread.

Signature

[154] Section 12.4.1.15
[155] Section 12.4.1.7

```
public
javax.realtime.device.RawFloat createRawFloat(RawMemoryRegion
region, long base, int count, int stride)
throws SecurityException, OffsetOutOfBoundsException,
SizeOutOfBoundsException, MemoryTypeConflictException,
UnsupportedRawMemoryRegionException
```

Parameters

base The starting physical address accessible through the returned instance.

count The number of memory elements accessible through the returned instance.

stride The distance to the next element as a mulitple of element size, where `1` means the elements are adjacent in memory.

Throws

IllegalArgumentException when `base` is negative, `count` is not greater than zero, or `stride` is less than one.

SecurityException when the caller does not have permissions to access the given memory region or the specified range of addresses.

OffsetOutOfBoundsException when `base` is invalid.

SizeOutOfBoundsException when the memory addressed by the object would extend into an invalid range of memory.

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

Returns

an object that implements `RawFloat`[156] and supports access to the specified range in the memory region.

createRawFloatReader(RawMemoryRegion, long, int, int)

Create an instance of a class that implements `RawFloatReader`[157] and accesses memory of `region` in the address range described by `base`, `stride`, and `count`. The actual extent of the memory addressed by the object is `stride` ∗ *size of RawFloatReader* ∗ `count`. The object is allocated in the current memory area of the calling thread.

Signature

```
public
```

[156] Section 12.4.1.7
[157] Section 12.4.1.8

```
javax.realtime.device.RawFloatReader createRawFloat-
Reader(RawMemoryRegion region, long base, int count, int stride)
throws SecurityException, OffsetOutOfBoundsException,
SizeOutOfBoundsException, MemoryTypeConflictException,
UnsupportedRawMemoryRegionException
```

Parameters

base The starting physical address accessible through the returned instance.

count The number of memory elements accessible through the returned instance.

stride The distance to the next element as a mulitple of element size, where `1` means the elements are adjacent in memory.

Throws

IllegalArgumentException when `base` is negative, `count` is not greater than zero, or `stride` is less than one.

SecurityException when the caller does not have permissions to access the given memory region or the specified range of addresses.

OffsetOutOfBoundsException when `base` is invalid.

SizeOutOfBoundsException when the memory addressed by the object would extend into an invalid range of memory.

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

Returns

an object that implements `RawFloatReader`[158] and supports access to the specified range in the memory region.

createRawFloatWriter(RawMemoryRegion, long, int, int)

Create an instance of a class that implements `RawFloatWriter`[159] and accesses memory of `region` in the address range described by `base`, `stride`, and `count`. The actual extent of the memory addressed by the object is `stride` ∗ *size of RawFloatWriter* ∗ `count`. The object is allocated in the current memory area of the calling thread.

Signature

```
public
javax.realtime.device.RawFloatWriter createRawFloatWri-
ter(RawMemoryRegion region, long base, int count, int stride)
```

[158] Section 12.4.1.8

[159] Section 12.4.1.9

`throws SecurityException, OffsetOutOfBoundsException, SizeOutOfBoundsException, MemoryTypeConflictException, UnsupportedRawMemoryRegionException`

Parameters

base The starting physical address accessible through the returned instance.

count The number of memory elements accessible through the returned instance.

stride The distance to the next element as a mulitple of element size, where `1` means the elements are adjacent in memory.

Throws

IllegalArgumentException when `base` is negative, `count` is not greater than zero, or `stride` is less than one.

SecurityException when the caller does not have permissions to access the given memory region or the specified range of addresses.

OffsetOutOfBoundsException when `base` is invalid.

SizeOutOfBoundsException when the memory addressed by the object would extend into an invalid range of memory.

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

Returns

an object that implements `RawFloatWriter`[160] and supports access to the specified range in the memory region.

createRawDouble(RawMemoryRegion, long, int, int)

Create an instance of a class that implements `RawDouble`[161] and accesses memory of `region` in the address range described by `base`, `stride`, and `count`. The actual extent of the memory addressed by the object is `stride` ∗ *size of RawDouble* ∗ `count`. The object is allocated in the current memory area of the calling thread.

Signature

```
public
javax.realtime.device.RawDouble createRawDouble(RawMemoryRegion
region, long base, int count, int stride)
throws SecurityException, OffsetOutOfBoundsException,
SizeOutOfBoundsException, MemoryTypeConflictException,
UnsupportedRawMemoryRegionException
```

[160] Section 12.4.1.9
[161] Section 12.4.1.4

Parameters

base The starting physical address accessible through the returned instance.

count The number of memory elements accessible through the returned instance.

stride The distance to the next element as a mulitple of element size, where `1` means the elements are adjacent in memory.

Throws

IllegalArgumentException when `base` is negative, `count` is not greater than zero, or `stride` is less than one.

SecurityException when the caller does not have permissions to access the given memory region or the specified range of addresses.

OffsetOutOfBoundsException when `base` is invalid.

SizeOutOfBoundsException when the memory addressed by the object would extend into an invalid range of memory.

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

Returns

an object that implements `RawDouble`[162] and supports access to the specified range in the memory region.

createRawDoubleReader(RawMemoryRegion, long, int, int)

Create an instance of a class that implements `RawDoubleReader`[163] and accesses memory of `region` in the address range described by `base`, `stride`, and `count`. The actual extent of the memory addressed by the object is `stride` ∗ *size of RawDoubleReader* ∗ `count`. The object is allocated in the current memory area of the calling thread.

Signature

```
public
javax.realtime.device.RawDoubleReader createRawDouble-
Reader(RawMemoryRegion region, long base, int count, int stride)
throws SecurityException, OffsetOutOfBoundsException,
SizeOutOfBoundsException, MemoryTypeConflictException,
UnsupportedRawMemoryRegionException
```

Parameters

[162] Section 12.4.1.4

[163] Section 12.4.1.5

base The starting physical address accessible through the returned instance.
count The number of memory elements accessible through the returned instance.
stride The distance to the next element as a mulitple of element size, where `1` means the elements are adjacent in memory.

Throws

IllegalArgumentException when `base` is negative, `count` is not greater than zero, or `stride` is less than one.
SecurityException when the caller does not have permissions to access the given memory region or the specified range of addresses.
OffsetOutOfBoundsException when `base` is invalid.
SizeOutOfBoundsException when the memory addressed by the object would extend into an invalid range of memory.
MemoryTypeConflictException when `base` does not point to memory that matches the type served by this factory.

Returns

an object that implements `RawDoubleReader`[164] and supports access to the specified range in the memory region.

createRawDoubleWriter(RawMemoryRegion, long, int, int)

Create an instance of a class that implements `RawDoubleWriter`[165] and accesses memory of `region` in the address range described by `base`, `stride`, and `count`. The actual extent of the memory addressed by the object is `stride` ∗ *size of RawDoubleWriter* ∗ `count`. The object is allocated in the current memory area of the calling thread.

Signature

```
public
javax.realtime.device.RawDoubleWriter createRawDoubleWri-
ter(RawMemoryRegion region, long base, int count, int stride)
throws SecurityException, OffsetOutOfBoundsException,
SizeOutOfBoundsException, MemoryTypeConflictException,
UnsupportedRawMemoryRegionException
```

Parameters

base The starting physical address accessible through the returned instance.

[164]Section 12.4.1.5
[165]Section 12.4.1.6

count The number of memory elements accessible through the returned instance.
stride The distance to the next element as a mulitple of element size, where `1` means the elements are adjacent in memory.

Throws

IllegalArgumentException when `base` is negative, `count` is not greater than zero, or `stride` is less than one.
SecurityException when the caller does not have permissions to access the given memory region or the specified range of addresses.
OffsetOutOfBoundsException when `base` is invalid.
SizeOutOfBoundsException when the memory addressed by the object would extend into an invalid range of memory.
MemoryTypeConflictException when `base` does not point to memory that matches the type served by this factory.

Returns

an object that implements `RawDoubleWriter`[166] and supports access to the specified range in the memory region.

12.4.3.6 RawMemoryRegion

RawMemoryRegion is a class for typing raw memory regions. It is returned by the `RawMemoryRegionFactory.getRegion`[167] methods of the raw memory region factory classes, and it is used with methods such as `RawMemoryFactory.createRawByte(RawMemoryRegion, long, int, int)`[168] and `RawMemoryFactory.createRawDouble(RawMemoryRegion, long, int, int)`[169] methods to identify the region from which the application wants to get an accessor instance.

This is distinct from the, similarly named `PhysicalMemoryName`[170] class, which is used with memory area classes, specifically `ImmortalPhysicalMemory`[171] and `LTPhysicalMemory`[172].

Available since RTSJ 2.0

Inheritance

[166] Section 12.4.1.6
[167] Section 12.4.1.17.1
[168] Section 12.4.3.5.3
[169] Section 12.4.3.5.3
[170] Section 15.3.1.1
[171] Section 11.4.3.3
[172] Section 11.4.3.5

java.lang.Object
  javax.realtime.device.RawMemoryRegion

12.4.3.6.1 Methods

---

getRegion(String)

Get a region type when it already exists or creates a new one.

Signature

```
public static
javax.realtime.device.RawMemoryRegion getRegion(String name)
```

Parameters

name of the region

Returns

the region type object.

isRawMemoryRegion(String)

Ask whether or not there is a memory region type of a given name.

Signature

```
public static
boolean isRawMemoryRegion(String name)
```

Parameters

name for which to search

Returns

`true` when there is one and `false` otherwise.

getName

Obtains the name of this region type.

Signature

```
public final
```

```
java.lang.String getName()
```

Returns

the region types name

toString

Gets a printable representation for a Region.

Signature

```
public final
java.lang.String toString()
```

Returns

the name of this memory region type.

12.5 Rationale

12.5.1 Raw Memory

Raw memory in the `RTSJ` refers to any memory in which only objects of primitive types can be stored; *Java objects or their references cannot be stored in raw memory.* 2.0 of specification provides two categories:

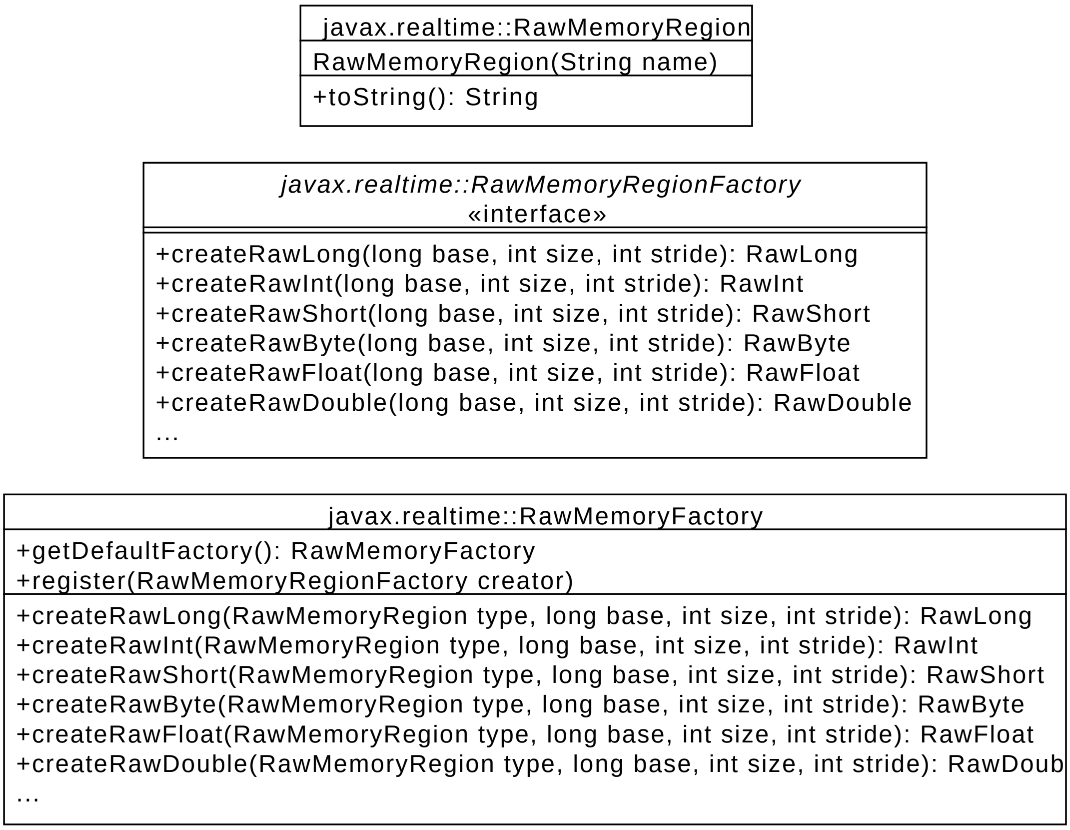
- memory that is used to access memory-mapped device registers, and
- logical memory that can be used to access port-based device registers.

Each of these categories of memory is represented by an instance of `RawMemoryRegion`. In addition, the application can define other regions outside these two, either for accessing devices registers in some other address space or for other purposes, such as emulating device access.

Java's primitive types are partitioned into two groups: integral (short, int, long, byte) and real (float, double) types, including arrays of each type. For integral types, individual interfaces are also defined to facilitate greater type security during access. Objects that support these interfaces are created by factory methods, which again have predefined interfaces. Such objects are called *accessor* objects as they encapsulates the access protocol to the raw memory.

Control over all these objects is managed by the `RawMemoryFactory` class that provides a set of static methods, as shown in Figure 12.5. There are two groups of methods, those that

Figure 12.5: Creating Raw Memory Accessors



- enable a factory to be registered, and
- request the creation of *accessor* object for a particular memory type at a particular address.

The latter consists of methods to create Java-primitive-type accessor objets, which will throw exceptions if the appropriate addresses are not on correct boundaries to enable the underlying machine instructions to be used without causing hardware exceptions (e.g., `createRawByteReader`).

As with interrupt handling, some realtime JVMs may not be able to support all of the memory categories. However, the expectation is that for all supported categories, they will also provide and register the associated factories for object creation.

For the case of `IO_PORT_MAPPED` raw memory, the accessor objects will need to arrange to execute the appropriate machine instructions to access the device registers.

Consider, the simple case where a device has a two device registers: a control/status register that is a 32 bits integer, and a data register that is a 64 bits long. The registers have been memory mapped to locations: 0x20 and 0x24 respectively. Assuming the realtime JVM has registered a factory for the `IO_PORT_MAPPED_REGION` raw memory name, then the following code will create the objects that facilitate the memory access

```
RawMemoryFactory factor = RawMemoryFactory.getDefaultFactory();
RawInt controlReg =
  factory.createRawInt(RawMemoryFactory.IO_PORT_MAPPED_REGION, 0x20);
RawLong dataReg =
  factory.createRawLong(RawMemoryFactory.IO_PORT_MAPPED_REGION, 0x24);
```

The above definitions reflect the structure of the actual registers. The JVM will check that the memory locations are on the correct boundaries and that they can be accessed without any hardware exceptions being generated. If they cannot, the create methods will throw an appropriate exceptions. If successfully created, all future access to the `controlReg` and `dataReg` will be exception free. The registers can be manipulated by calling the appropriate methods, as in the following example.

```
dataReg.put(l);
    // where l is of type long and is data to be sent to the device
controlReg.put(i);
    // where i is of type int and is the command to the device
```

In the general case, programmers themselves may create their own memory categories and provide associated factories (that may use the implementation-defined factories). These factories are written in Java and are, therefore, constrained by what the language allows them to do. Typically, they will use the JVM-supplied raw memory types to facilitate access to a device's external memory.

Open issue: Andy: is the following still true? **End of open issue** In addition to the above facilities, the `RTSJ` also supports the notion of removable memory. When this memory is inserted or removed, an asynchronous event can be set up to fire, thereby alerting the application that the device has become active. Of course, any removable memory has to be treated with extreme caution by the realtime JVM. Hence, the `RTSJ` allows it only to be accessed as a raw memory device. An example of these latter facilities will be given in Section 12.5.3.

12.5.1.1 Direct memory access

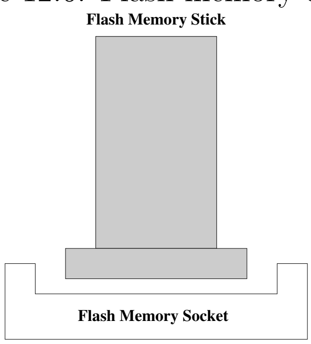
DMA requires access to memory out side of the heap. It is often crucial for performance in embedded systems; however, it does cause problems both from a realtime analysis perspective and from a JVM-implementation perspective. The latter is the primary concern here.

There are a few crucial points to note about DMA and the `RTSJ`.

- The `RTSJ` does not address issues of persistent objects; so the input and output of Java objects to devices (other than by using the Java serialization mechanism) is not supported.
- The `RTSJ` requires that `RTSJ` programs can be compiled by regular Java compilers. Different bytecode compilers (and their supporting JVM) use different representation for objects. Java arrays (even of primitive types) are objects, and the data they contain might not be stored in contiguous memory.
- The package `java.nio.channels` provides a mechanism for I/O that was not specifically designed for DMA, but provides an applicable pattern for it.

For these reasons, without explicit knowledge of the compiler and JVM, allowing any DMA into any `RTSJ` memory area is a very dangerous action; therefore, the `RTSJ` provides some special support for DMA. Unfortunately, it would be difficult to find a general pattern to fit all DMA controllers. With raw memory and raw byte buffers, one could construct a higher level API that would cover most DMA controllers, but there will always odd cases that would still not fit the general pattern, especially for embedded systems. For this reason, only this low level API is provided.

Figure 12.6: Flash memory device



12.5.2 Interrupt Handling

Handling interrupts is a necessary part of many embedded systems. Interrupt handlers have traditionally been implemented in assembler code or C. With the growing popularity of high-level concurrent languages, there has been interest in better integration between the interrupt handling code and the application. Ada, for example, allows a "protected" procedure to be called directly from an interrupt [3].

Regehr [7] defines the terms used for the core components of interrupts and their handlers as follows.

- *Interrupt*—a hardware supported asynchronous transfer of control mechanism initiated by an event external to the processor. Control of the processor is transferred through an interrupt vector.
- *Interrupt vector*—a dedicated (or configurable) location that specifies the location of an interrupt handler.
- *Interrupt handler*—code that is reachable from the interrupt vector.
- *An interrupt controller*—a peripheral device that manages interrupts for the processor.

He further identifies the following problems with programming interrupt-driven software on single processors:

- Stack overflow—the difficulty determining how much call-chain stack is required to handle an interrupt. The problem is compounded if the stack is borrowed from the currently executing thread or process.
- Interrupt overload—the problem of ensuring that non-interrupt driven processing is not swamped by unexpected or misbehaving interrupts.
- Real-time analysis—the need to have appropriate schedulability analysis models to bound the impact of interrupt handlers.

The problems above are accentuated in multiprocessor systems where interrupts can be handled globally. Fortunately, many multiprocessor systems allow interrupts to be bound to particular processors. For example, the ARM Cortex A9-MPCore supports the Arm Generic Interrupt Controller[173]. This enables a target list of CPUs to be specified for each hardware interrupt. Software generated interrupts can also be sent to the list or set up to be delivered to all but the requesting CPU or only the requesting CPU.

Regehr's problems are all generic and can be solved irrespective of the language used to implement the handlers. In general they can be addressed by a combination of techniques.

- Stack overflow—static analysis techniques can usually be used to determine the worst-case stack usage of all interrupt handlers. If stack is borrowed from the executing thread then this amount must be added to the worst-case stack usage of all threads.
- Interrupt overload—this is typically managed by aperiodic server technology in combination with interrupt masking (see Section 13.6 of [3]).
- Real-time analysis—again this can be catered for in modern schedulability analysis techniques, such as response-time analysis (see Section 14.6 of [3]).

From a `RTSJ` perspective, the following distinctions are useful

- The *first-level interrupt handlers* are the code that the platform executes in response to the hardware interrupts (or traps). A first-level interrupt is assumed to be executed at an execution eligibility (priority) and by a processor dictated

[173] See http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.ddi0375a/Cegbfjhf.html

by the underlying platform (which may be controllable at the platform level). On some `RTSJ` implementations it will not be possible to write Java code for these handlers. Implementations that do enable Java-level handlers may restrict the code that can be written. For example, the handler code should not suspend itself or throw unhandled exceptions. The `RTSJ` 2.0 optional `InterruptServiceRoutine` class supports first level interrupt handling.

- The *external event handler* is the code that the JVM executes as a result of being notified that an external event (be it an operating system signal, an ISR or some other program) is targeted at the `RTSJ` application. The programmer should be able to specify the processor affinity and execution eligibility of this code. In `RTSJ` 2.0, all external events are represented by instances of the `Happening` interface. Every happening has an associated dispatcher which is responsible for the initial response to an occurrence of the event.
- A happening dispatcher is able to find one or more associated `RTSJ` asynchronous events and fire them. This then releases the associated asynchronous event handlers.

12.5.3 An Illustrative Example

Consider an embedded system that has a simple flash memory device that supports a single type of removable flash memory stick, as illustrated in Figure 12.6.

When the memory stick is inserted or removed, an interrupt is generated. This interrupt is known to the realtime JVM. The interrupt is also generated when operations requested on the device are completed. For simplicity, it is assumed that realtime JVM has mapped this interrupt to an external happening called `FlashHappening` with a default happening dispatcher.

The example illustrates how

1. a programmer can use the `RTSJ` facilities to write a device handler,
2. a factory class can be constructed and how the accessor objects police the access,
3. removable memory is handled.

The flash memory device is accessed via several associated registers, which are shown in Table 12.1. These have all been memory mapped to the indicated locations.

12.5.3.1 Software architecture

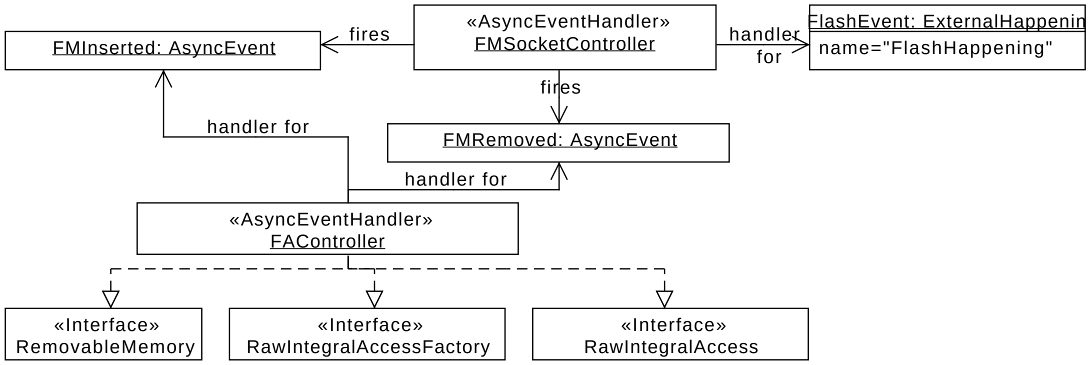
There are many ways in which the software architecture for the example could be constructed. Here, for simplicity of representation, an architecture is chosen with a minimal number of classes. It is illustrated in Figure 12.7. There are three key components.

Table 12.1: Device registers

Register	Location	Bit Positions	Values
Command	0x20	0	0 = Disable device, 1 = Enable device
		4	0 = Disable interrupts, 1 = Enable interrupts
		5-8	1 = Read byte, 2 = Write byte
			3 = Read short, 4 = Write short
			5 = Read int, 6 = Write int
			7 = Read long, 8 = Write long
		9	0 = DMA Read, 1 = DMA
		31-63	Offset into flash memory
Data	0x28	0-63	Simple data or memory address if DMA
Length	0x30	0-31	Length of data transfer
Status	0x38	0	1 = Device enabled
		3	1 = Interrupts enabled
		4	1 = Device in error
		5	1 = Transfer complete
		6	1 = Memory stick present
			0 = Memory stick absent
		7	1 = Memory stick inserted
		8	0 = Memory stick removed

- `FlashHappening`—This is the external happening that is associated with the flash device's interrupt. The `RTSJ` will provide a default dispatcher, which will fire the asynchronous event when the interrupt occurs.
- `FMSocketController`—This is the object that encapsulates the access to the flash memory device. In essence, it is the device driver; it is also the handler for the `FlashHappening` and is responsibly for firing the `FMInserted` and `FMRemoved` asynchronous events.
- `FAController`—This is the object that controls access to the flash memory, it
  - acts as the factory for the creating objects that will facilitate access to the flash memory itself (using the mechanisms provided by the `FMSocketController`),
  - is the asynchronous event handler that responds to the firing of the `FMInserted` and `FMRemoved` asynchronous events, and
  - also acts as the accessor object for the memory.

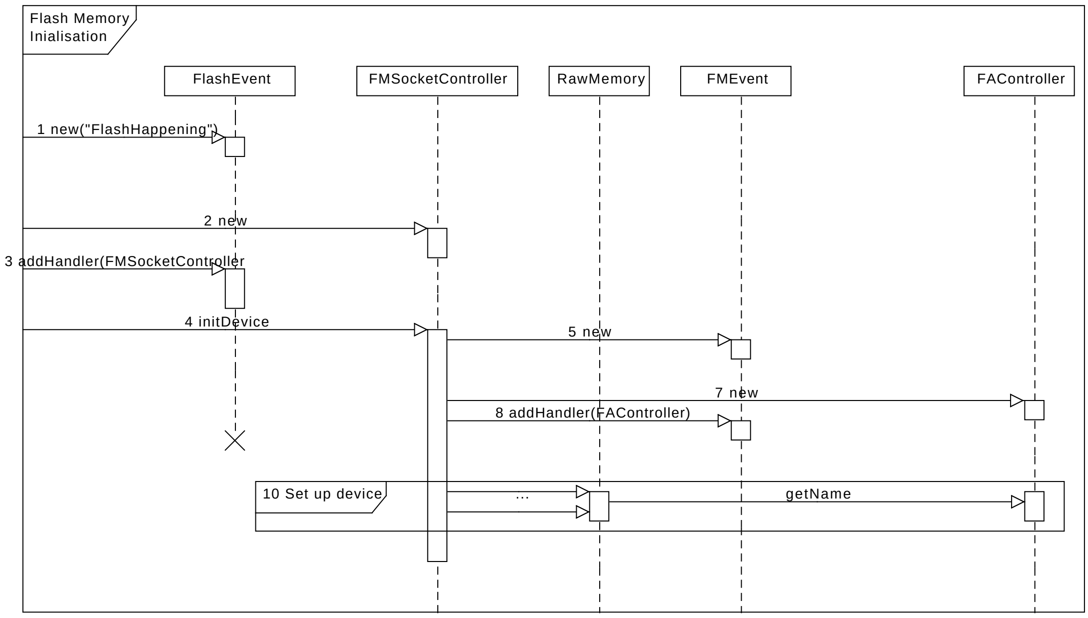
Figure 12.7: Flash memory classes



12.5.3.2 Device initialization

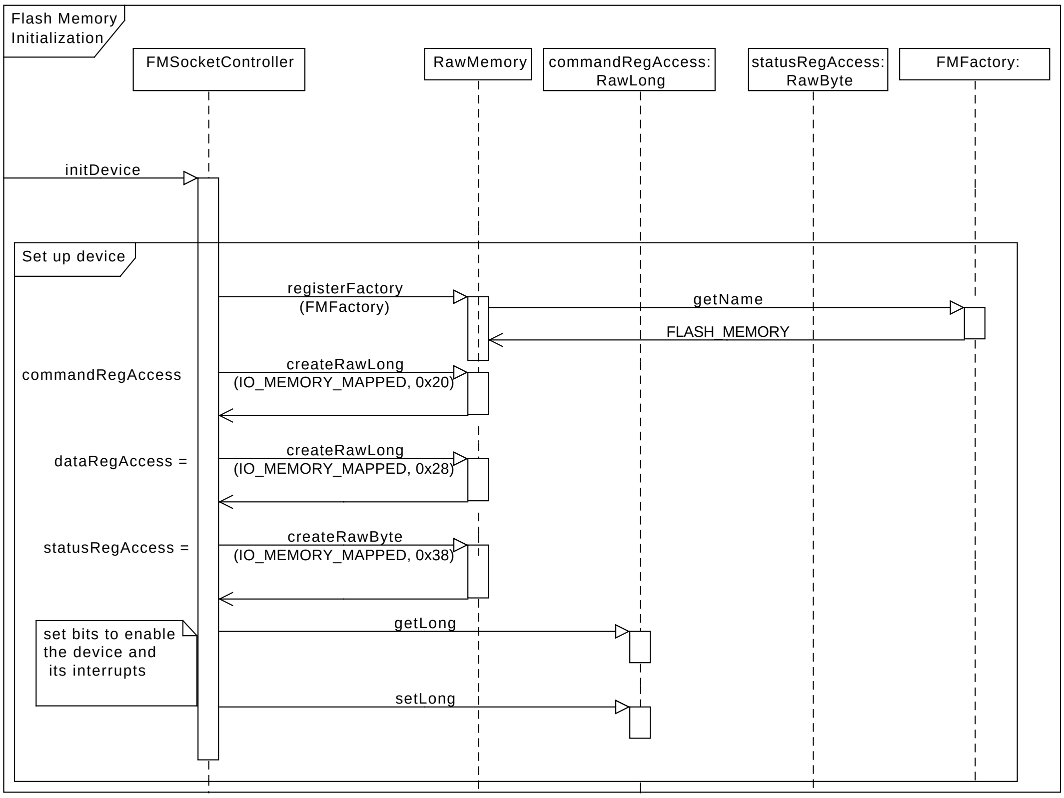
Figure 12.8 shows the sequence of operations that the program must perform to initialize the flash memory device. The main steps are as follows.

Figure 12.8: Sequence diagram showing initialization operations



1 The external happening (`FlashEvent`) associated with the flash happening must be created.

Figure 12.9: Sequence diagrams showing operations to initialize the hardware device

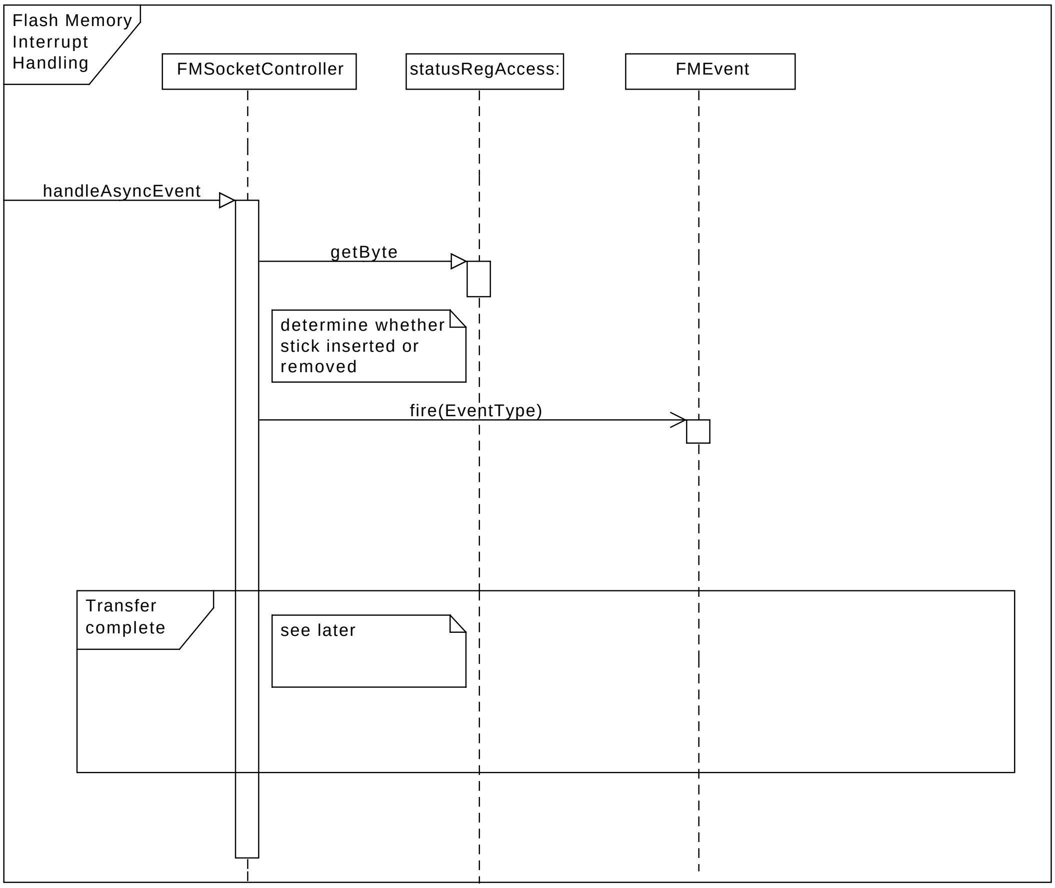


- 2-3 The (`FMSocketController`) object is created and added as a handler for `FlashEvent`.
- 4 An initialization method is called (`initDevice`) to perform all the operations necessary to configure the infrastructure and initialize the hardware device.
- 5-6 Two new asynchronous events are created to represent insertion and removal of the flash memory stick.
- 7-9 The `FAController` class is created. It is added as the handler for the two events created in steps 5 and 6.
- 10 Setting up the device and registering the factory is shown in detail in Figure 12.9. It involves: registering the `FAController` object via the static methods in the `RawMemory` class, and creating and using the JVM-supplied factory to access the memory-mapped I/O registers.

12.5.3.3 Responding to external happenings

In the example, interrupts are handled by the JVM, which turns them into an external happening. The application code that indirectly responds to the happening

Figure 12.10: The `FMSocketController.handleAsync` method



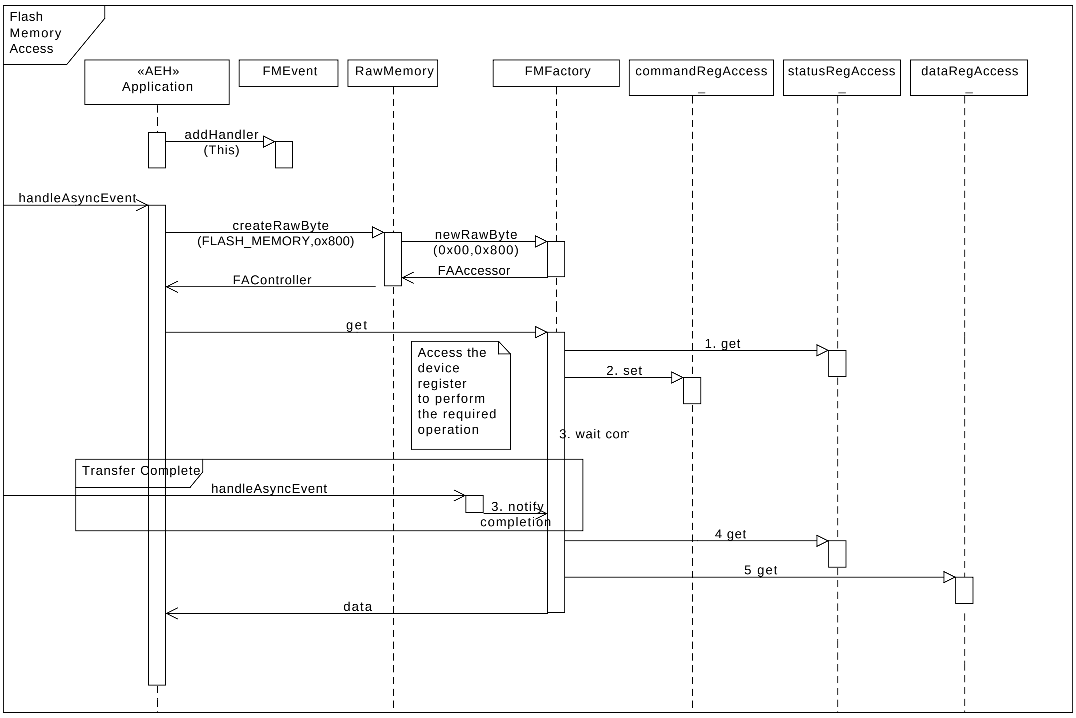
is provided in the `handleAsyncEvent` method in the `FMSocketController` object. Figure 12.10 illustrates the approach. In this example, the actions in response to the *memory stick inserted* and *memory stick removed* flash events is simply shown as the execution of the `FMInserted` and `FMRemoved` handlers. These will inform the application. The memory accessor classes themselves will ensure that the stick is present when performing the required application accesses.

12.5.3.4 Access to the flash controller's device registers

Figure 12.11 shows the sequence of events that the application follows. First it must register a handler with the `FMInserted` asynchronous event. Here, the application itself is an asynchronous event handler. When this is released, the memory has been inserted.

In this simple example, the application simply reads a byte from an offset within the memory stick. It, therefore, creates an accessor to access the data. When

Figure 12.11: Application usage



this has been returned (it is the `FAController` itself), the application can now call the `get` method (called *FA* `get`, in the following, for clarity). This method must implement the sequence of raw memory access on the device's registers to perform the operation. In Figure 12.11, they are as follows.

1. *FA* `get` calls the `get` method of the status register's accessor object. This can check to make sure that the flash memory is present (bit 6, as shown in Table 12.1). If it is not, an exception can be thrown.
2. Assuming the memory is present, it then sets the control register with the offset required (bits 31–63, as shown in Table 12.1) and sets the read byte request bit (bits 5-8, as shown in Table 12.1).
3. The *FA* `get` method must then wait for indication that the requested operation has been completed by the device. This is detected by the `handleAsyncEvent` method of the `FMController`, which performs the necessary notify.
4. Once notified of completion, the *FA* `get` method, again reads the status register to make sure there were no errors on the device (bit 4 in Table 12.1) and that the memory is still present
5. The *FA* `get` then reads the data register to get the requested data, which it returns.

Chapter 13

System and Options

13.1 Overview

Implementations of this specification run on many operating systems and this specification itself supports several variants, therefore a means of querying and handling this variation is required. For instance, though many realtime operating systems support the POSIX standard, many do not as well, Even the ones that do vary in their degree of compliance. POSIX provides a well defined means of signalling other processes and receiving signals from them, therefore one would like to be able to use this facility when available. The specification defines classes to help manage these differences providing the following:

- a class that contains operations and semantics that affect the entire system;
- the security semantics required by the additional features in the entirety of this specification, which are additional to those required by implementations of the Java Language Specification; and
- a common idiom for binding POSIX signals to instances of `AsyncEventHandler` when POSIX signals are available on the underlying platform.

The `RealtimeSecurity` class provides security primarily for physical memory access.

13.2 Semantics

There are four classes with semantics that do not fall into other categories: `RealtimeSystem`, `RealtimeSecurity`, `POISIXSignal`, and `POSIXRealtimeSignal`. Their overall semantics is detailed below. Afterwards, semantics that apply to methods, constructors, and fields of theses classes are provided as well.

13.2.0.5 RealtimeSystem

RealtimeSystem is a required class, which provides basic information about the RTSJ extensions supported by the system. Via this class, a program can query the default monitor policy, the realtime security manager, and other realtime properties of the system. Starting from version 2, a program can also ask what modules are supported. Determining the current version is supported by a system property. If an application call the method, `System.getProperty(‘‘javax.realtime.version’’)`, the return value will be a string of the form, “x.y.z”. Where ’x’ is the major version number and ’y’ and ’z’ are minor version numbers. These version numbers state to which version of the `RTSJ` the underlying implementation claims conformance. The first release of the `RTSJ`, dated 11/2001, was numbered 1.0.0. A release conforming to this version of the specification should return the string `"2.0.0"`.

13.2.0.6 RealtimeSecurity

The RealtimeSecurity class controls access to key realtime features. Particularly critical is access to memory outside the heap. Core RTSJ features also have security checks. This should enable an application to restrict the use of the RTSJ, particularly for dynamically loaded code. Detailed information is provided in the class documentation below.

13.2.0.7 POSIX Signals

The POSIXSignal class represents `POSIX` signals and is required on platforms that provide `POSIX` signals. As a `Happening`, it is a subclass of `AsyncEvent` and implements `ActiveEvent`. Unlike `Happening`, it cannot be instantiated by the user. Instead, an instance exists for each POSIX signal defined on the system. They can be retrieved either by name or number using the POSIXSignal.get(int) and POSIXSignal.get(String) methods.

13.2.0.8 POSIX Realtime Signals

The POSIXRealtimeSignal class represents POSIX realtime events. It is also implements `ActiveEvent`, but is a subclass of `AsyncLongEvent`, so that it can pass the data sent with its signal. As with `POSIXSignal`, it cannot be instantiated by the user, rather an instance exists for each POSIX signal defined on the system, which can be retrieves either by name or number using the POSIXRealtimeSignal.get(int) and POSIXRealtimeSignal.get(String) methods.

13.3 Package javax.realtime

13.3.1 Classes

13.3.1.1 GarbageCollector

The system shall provide dynamic and static information characterizing the temporal behavior and imposed overhead of any garbage collection algorithm provided by the system. This information shall be made available to applications via methods on subclasses of `GarbageCollector`. Implementations are allowed to provide any set of methods in subclasses as long as the temporal behavior and overhead are sufficiently categorized. The implementations are also required to fully document the subclasses.

A reference to the garbage collector responsible for heap memory is available from `RealtimeSystem.currentGC()`[1].

Inheritance

java.lang.Object
  javax.realtime.GarbageCollector

13.3.1.1.1 Methods

getPreemptionLatency

Preemption latency is a measure of the maximum time a schedulable object may have to wait for the collector to reach a preemption-safe point.

Instances of `NoHeapRealtimeThread`[2] and async event handlers with the no-heap option preempt garbage collection immediately, but other schedulables must wait until the collector reaches a preemption-safe point. For many garbage collectors the only preemption safe point is at the end of garbage collection, but an implementation of the garbage collector could permit a schedulable to preempt garbage collection before it completes. The `getPreemptionLatency` method gives such a garbage collector a way to report the worst-case interval between release of a schedulable during garbage collection, and the time the schedulable starts execution or gains full access to heap memory, whichever comes later.

[1] Section 13.3.1.7.2
[2] Section 15.3.2.1

Signature

```
public abstract
javax.realtime.RelativeTime getPreemptionLatency()
```

Returns

The worst-case preemption latency of the garbage collection algorithm represented by `this`. The returned object is allocated in the current allocation context. When there is no constant that bounds garbage collector preemption latency, this method shall return a relative time with `Long.MAX_VALUE` milliseconds. The number of nanoseconds in this special value is unspecified.

13.3.1.2 POSIXRealtimeSignal

A `ActiveEvent`[3] subclass for defining a POSIX realtime signal.
Available since RTSJ 2.0

Inheritance

java.lang.Object
  javax.realtime.AbstractAsyncEvent
    javax.realtime.AsyncLongEvent
      javax.realtime.POSIXRealtimeSignal

Interfaces

ActiveEvent

13.3.1.2.1 Methods

---

get(String)

Get the registered realtime signal with the given name.

Signature

```
public static
javax.realtime.POSIXRealtimeSignal get(String name)
```

[3]Section 8.4.1.1

Parameters
: *name* of the signal to get.

Returns
: the registered signal with name or `null`.

getId(String)

Get the ID of a registered signal.

Signature

```
public static
int getId(String name)
```

Parameters
: *name* of the signal for which to search

Returns
: the ID of the signal named by `name`

get(int)

Get the realtime signal corresponding to a given id.

Signature

```
public static
javax.realtime.POSIXRealtimeSignal get(int id)
```

Parameters
: *id* of a registered signal

Returns
: the signal corresponding to id.

isPOSIXRealtimeSignal(String)

Determine if a signal with a given name is registered.

Signature

```
public static
boolean isPOSIXRealtimeSignal(String name)
```

Parameters

name of the signal

Returns

true when a signal with the given name is registered

trigger(int, long)

Release the manager for the Realtime Signal identified by the given integer. The id range for POSiX Signals is distinct from that of other types of `ActiveEvent`[4]. This method is provided for simulating an occurrence of a realtime POSIX signal.

Signature

```
public static
void trigger(int id, long value)
```

Parameters

id of a registered signal

value passed from the signaler.

getId

Get the name of this realtime signal.

Signature

```
public
int getId()
```

Returns

its name.

getName

Get the name of this signal.

Signature

```
public final
java.lang.String getName()
```

Returns

the name of this signal.

[4]Section 8.4.1.1

isActive

Determine the activation state of this signal, i.e., it has been started.

Signature

```
public
boolean isActive()
```

Returns

`true` when active, `false` otherwise.

isRunning

Determine the firing state (releasing or skipping) of this signal, i.e., it is active and enabled.

Signature

```
public
boolean isRunning()
```

Returns

`true` when releasing, `false` when skipping.

start

Start this `POSIXRealtimeSignal`, i.e., change to a running state. A running realtime signal is a source of activation when in a scoped memory and is a member of the root set when in the heap. A running realtime signal can be triggered.
See Section stop()

Signature

```
public final synchronized
void start()
throws IllegalStateException
```

Throws

IllegalStateException when this `POSIXRealtimeSignal` has already been started.

start(boolean)

Start this `POSIXRealtimeSignal`, i.e., change to a running state. A running realtime signal is a source of activation when in a scoped memory and is a member of the root set when in the heap. A running realtime signal can be triggered.
See Section stop()

Signature

```
public final synchronized
void start(boolean disabled)
throws IllegalStateException
```

Parameters

disabled true for starting in a disabled state.

Throws

IllegalStateException when this `POSIXRealtimeSignal` has already been started.

stop

Stop this `POSIXRealtimeSignal`. A stopped realtime signal ceases to be a source of activation and no longer cause any AE attached to it to be a source of activation.

Signature

```
public final
boolean stop()
throws IllegalStateException
```

Throws

IllegalStateException when this `POSIXRealtimeSignal` is not running.

Returns

`true` when `this` was *enabled* and `false` otherwise.

getNextValue

Signature

```
public
long getNextValue()
```

Returns

the value of the next signal

send(long)

Send this signal to another process

Signature

```
public native
boolean send(long pid)
```

Parameters

pid of the process to which to send the signal

Returns

true when signal can be sent, otherwise false.

13.3.1.3 POSIXRealtimeSignalDispatcher

Provides a means of dispatching a set of `POSIXRealtimeSignal`[5]s. An application can provide its own dispatcher, providing the priority for the internal dispatching thread. This dispatching thread calls process() each time the signal is triggered.
Available since RTSJ 2.0

Inheritance

java.lang.Object
  javax.realtime.ActiveEventDispatcher
    javax.realtime.POSIXRealtimeSignalDispatcher

13.3.1.3.1 Constructors

POSIXRealtimeSignalDispatcher(SchedulingParameters)

Create a new dispatcher, whose dispatching thread runs with the given scheduling parameters.

[5]Section 13.3.1.2

Signature

```
public
  POSIXRealtimeSignalDispatcher(SchedulingParameters schedule)
```

Parameters

schedule give the parameters for scheduling this dispatcher

13.3.1.3.2 Methods

register(POSIXRealtimeSignal)

Register `signal` with this dispatcher.

Signature

```
public
void register(POSIXRealtimeSignal signal)
throws RegistrationException, IllegalStateException,
IllegalArgumentException
```

Parameters

signal to register

Throws

RegistrationException when `signal` is already registered.
IllegalStateException when this object has been destroyed.
IllegalArgumentException when `signal` is not stopped.

deregister(POSIXRealtimeSignal)

Deregister the `signal` form this dispatcher.

Signature

```
public
void deregister(POSIXRealtimeSignal signal)
```

`throws DeregistrationException, IllegalStateException, IllegalArgumentException`

Parameters

signal to unregister

Throws

DeregistrationException when `signal` is not already registered.
IllegalStateException when this object has been destroyed.
IllegalArgumentException when `signal` is not stopped.

destroy

Release all reasources thereby making the dispatcher unusable.

Signature

```
public
void destroy()
throws IllegalStateException
```

Throws

IllegalStateException when called on a dispatcher that has one or more registered `POSIXRealtimeSignal`[6] objects.

13.3.1.4 POSIXSignal

A `ActiveEvent`[7] subclass for defining a POSIX signal.
Available since RTSJ 2.0

Inheritance

java.lang.Object
  javax.realtime.AbstractAsyncEvent
    javax.realtime.AsyncEvent
      javax.realtime.POSIXSignal

Interfaces

ActiveEvent

[6]Section 13.3.1.2
[7]Section 8.4.1.1

13.3.1.4.1 Fields

MAX_NUM_SIGNALS

`public static final MAX_NUM_SIGNALS`
this number of signals can be processed.

13.3.1.4.2 Methods

isPOSIXSignal(String)

Determine if a signal with a given name is registered.

Signature

```
public static
boolean isPOSIXSignal(String name)
```

Parameters

name of the signal

Returns

true when a signal with the given name is registered

get(String)

Get the registered signal with the given name.

Signature

```
public static
javax.realtime.POSIXSignal get(String name)
```

Parameters

name of the signal to get.

Returns

the registered signal with name or `null`.

getId(String)

Get the ID of a registered signal.

Signature

```
public static
int getId(String name)
```

Parameters

name of the signal for which to search

Returns

the ID of the signal named by `name`

get(int)

Get the signal corresponding to a given id.

Signature

```
public static
javax.realtime.POSIXSignal get(int id)
```

Parameters

id of a registered signal

Returns

the signal corresponding to id or `null`.

trigger(int)

Release the manager for the Signal identified by the given integer. The id range for Signals is distinct from that of other types of `ActiveEvent`[8]. This method is provided for simulating occurrences of a POSIX signal.

Signature

```
public static
void trigger(int id)
```

Parameters

id of a registered signal

[8]Section 8.4.1.1

getId

Get the number of this signal.

Signature

```
public
int getId()
```

Returns

the signal number

getName

Get the name of this signal.

Signature

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

Returns

the name of this signal.

isActive

Determine the activation state of this signal, i.e., it has been started.

Signature

```
public
boolean isActive()
```

Returns

`true` when active, `false` otherwise.

isRunning

Determine the firing state (releasing or skipping) of this signal, i.e., it is active and enabled.

Signature

```
public
boolean isRunning()
```

Returns

`true` when releasing, `false` when skipping.

start

Start this `POSIXSignal`, i.e., change to a running state. A running signal is a source of activation when in a scoped memory and is a member of the root set when in the heap. A running signal can be triggered.
See Section stop()

Signature

```
public
void start()
throws IllegalStateException
```

Throws

IllegalStateException when this `POSIXSignal` has already been started.

start(boolean)

Start this `POSIXSignal`, i.e., change to a running state. A running signal is a source of activation when in a scoped memory and is a member of the root set when in the heap. A running signal can be triggered.
See Section stop()

Signature

```
public
void start(boolean disabled)
throws IllegalStateException
```

Parameters

disabled true for starting in a disabled state.

Throws

IllegalStateException when this `POSIXSignal` has already been started.

stop

Stop this `POSIXSignal`. A stopped signal ceases to be a source of activation and no longer cause any AE attached to it to be a source of activation.

Signature

```
public
boolean stop()
throws IllegalStateException
```

Throws

IllegalStateException when this `POSIXSignal` is not running.

Returns

`true` when `this` was *enabled* and `false` otherwise.

send(long)

Send this signal to another process.

Signature

```
public native
boolean send(long pid)
```

Parameters

pid of the process to which to send the signal

Returns

true when signal can be sent, otherwise false.

13.3.1.5 POSIXSignalDispatcher

Provides a means of dispatching a set of `POSIXSignal`[9]s. An application can provide its own dispatcher, providing the priority for the internal dispatching thread. This dispatching thread calls process() each time the signal is triggered.

Inheritance

java.lang.Object
  javax.realtime.ActiveEventDispatcher
    javax.realtime.POSIXSignalDispatcher

[9]Section 13.3.1.4

13.3.1.5.1 Constructors

POSIXSignalDispatcher(SchedulingParameters)

Create a new dispatcher, whose dispatching thread runs with the given scheduling parameters.

Signature

```
public
  POSIXSignalDispatcher(SchedulingParameters scheduling)
```

Parameters

scheduling give the parameters for scheduling this dispatcher

13.3.1.5.2 Methods

register(POSIXSignal)

Register a POSIX signal with this dispatcher.

Signature

```
public synchronized
void register(POSIXSignal signal)
throws RegistrationException, IllegalStateException,
IllegalArgumentException
```

Parameters

signal to register

Throws

RegistrationException when `signal` is already registered.
IllegalStateException when this object has been destroyed.
IllegalArgumentException when `signal` is not stopped.

deregister(POSIXSignal)

Deregister a POSIX Signal form this dispatcher. (This is a really naive implementation.)

Signature

```
public synchronized
void deregister(POSIXSignal signal)
throws DeregistrationException, IllegalStateException,
IllegalArgumentException
```

Parameters

signal to deregister

Throws

DeregistrationException when `signal` not is already registered.
IllegalStateException when this object has been destroyed.
IllegalArgumentException when `signal` is not stopped.

destroy

Release all reasources thereby making the dispatcher unusable.

Signature

```
public
void destroy()
throws IllegalStateException
```

Throws

IllegalStateException when called on a dispatcher that has one or more registered `POSIXSignal`[10] objects.

13.3.1.6 RealtimeSecurity

Security policy object for realtime specific issues. Primarily used to control access to physical memory.

Security requirements are generally application-specific. Every implementation shall have a default `RealtimeSecurity` instance, and a way to install a replacement

[10]Section 13.3.1.4

at run-time, `RealtimeSystem.setSecurityManager`[11]. The default security is minimal. All security managers should prevent access to JVM internal data and the Java heap; additional protection is implementation-specific and must be documented.

Inheritance

java.lang.Object
  javax.realtime.RealtimeSecurity

13.3.1.6.1 Constructors

RealtimeSecurity

Create an `RealtimeSecurity` object.

Signature

```
public
  RealtimeSecurity()
```

13.3.1.6.2 Methods

---

checkAccessPhysical

Check whether the application is allowed to access physical memory.

Signature

```
public
void checkAccessPhysical()
throws SecurityException
```

Throws

SecurityException The application doesn't have permission to access physical memory.

[11] Section 13.3.1.7.2

checkAccessPhysicalRange(long, long)

Checks whether the application is allowed to access physical memory within the specified range.

Signature

```
public
void checkAccessPhysicalRange(long base, long size)
throws SecurityException
```

Parameters

base The beginning of the address range.

size The size of the address range.

Throws

SecurityException The application doesn't have permission to access the memory in the given range.

checkSetFilter

Checks whether the application is allowed to register `PhysicalMemoryTypeFilter`[12] objects with the `PhysicalMemoryManager`[13].

Signature

```
public
void checkSetFilter()
throws SecurityException
```

Throws

SecurityException The application doesn't have permission to register filter objects.

checkSetMonitorControl(MonitorControl)

Checks whether the application is allowed to set the default monitor control policy.

Available since RTSJ 1.0.1

[12]Section 15.3.1.2

[13]Section 15.3.2.3

Signature

```
public
void checkSetMonitorControl(MonitorControl policy)
throws SecurityException
```

Parameters

policy The new policy

Throws

SecurityException when the application doesn't have permission to change the default monitor control policy to `policy`.

checkAEHSetDaemon

Checks whether the application is allowed to set the daemon status of an AEH.
Available since RTSJ 1.0.1

Signature

```
public
void checkAEHSetDaemon()
throws SecurityException
```

Throws

SecurityException when the application is not permitted to alter the daemon status.

checkSetScheduler

Checks whether the application is allowed to set the scheduler.

Signature

```
public
void checkSetScheduler()
throws SecurityException
```

Throws

SecurityException The application doesn't have permission to set the scheduler.

checkCreateRealtimeThread

Check if an application may create a realtime thread.
Available since RTSJ 2.0

Signature

```
public
void checkCreateRealtimeThread()
throws SecurityException
```

Throws

SecurityException when not allowed

checkCreateTimer

Check if an application may create a Timer.
Available since RTSJ 2.0

Signature

```
public
void checkCreateTimer()
throws SecurityException
```

Throws

SecurityException when not allowed.

13.3.1.7 RealtimeSystem

`RealtimeSystem` provides a means for tuning the behavior of the implementation by specifying parameters such as the maximum number of locks that can be in use concurrently, and the monitor control policy. In addition, `RealtimeSystem` provides a mechanism for obtaining access to the security manager, garbage collector, and scheduler, to query or set parameters.

Open issue: Should there be flags to indicate which options are implemented?
End of open issue

Inheritance

java.lang.Object
  javax.realtime.RealtimeSystem

13.3.1.7.1 Fields

BIG_ENDIAN

`public static final BIG_ENDIAN`

Value indicating that the highest order byte of a bit word is stored at the lowest byte address: the int 0x0A0B0C0D is stored in the byte sequence 0x0A, 0x0B, 0x0C, 0x0D. and the long 0x0102030405060708 is stored in the sequence 0x01, 0x02, 0x03, 0x04, 0x05, 0x06, 0x07, 0x08.

LITTLE_ENDIAN

`public static final LITTLE_ENDIAN`

Value indicating that the lowest order byte of a word is stored at the lowest byte address: the int 0x0A0B0C0D is stored in the byte sequence 0x0D, 0x0C, 0x0B, 0x0A and the long 0x0102030405060708 is stored in the sequence 0x08, 0x07, 0x06, 0x05, 0x04, 0x03, 0x02, 0x01.

PDP_ENDIAN

`public static final PDP_ENDIAN`

Value indicating a mixed endian mode used by among others the PDP-11: the int 0x0A0B0C0D is stored in the byte sequence 0x0B, 0x0A, 0x0D, 0x0C, and the long 0x0102030405060708 is stored in the sequence 0x03, 0x04, 0x01, 0x02, 0x07, 0x08, 0x05, 0x06.

BYTE_ORDER

`public static final BYTE_ORDER`

The byte ordering of the underlying hardware.

13.3.1.7.2 Methods

---

currentGC

Return a reference to the currently active garbage collector for the heap.

Signature

```
public static
javax.realtime.GarbageCollector currentGC()
```

Returns

A `GarbageCollector`[14] object which is the current collector collecting objects on the traditional Java heap.

getConcurrentLocksUsed

Gets the maximum number of locks that have been used concurrently. This value can be used for tuning the concurrent locks parameter, which is used as a hint by systems that use a monitor cache.

Signature

```
public static
int getConcurrentLocksUsed()
```

Returns

An integer whose value is the maximum number of locks that have been used concurrently. When the number of concurrent locks is not tracked by the implementation, return -1. Note that when the number of concurrent locks is not tracked, the number of available concurrent locks is effectively unlimited.

getMaximumConcurrentLocks

Gets the maximum number of locks that can be used concurrently without incurring an execution time increase as set by the `setMaximumConcurrentLocks()` methods.

Note: Any relationship between this method and `setMaximumConcurrentLocks` is implementation-specific. This method returns the actual maximum number of concurrent locks the platform can currently support, or `Integer.MAX_VALUE` when there is no maximum. The `setMaximumConcurrentLocks` method give the implementation a hint as to the maximum number of concurrent locks it should expect.

[14] Section 13.3.1.1

Signature

```
public static
int getMaximumConcurrentLocks()
```

Returns

An integer whose value is the maximum number of locks that can be in simultaneous use.

getSecurityManager

Gets a reference to the security manager used to control access to realtime system features such as access to physical memory.

Signature

```
public static
javax.realtime.RealtimeSecurity getSecurityManager()
```

Returns

A `RealtimeSecurity`[15] object representing the default realtime security manager.

setMaximumConcurrentLocks(int)

Sets the anticipated maximum number of locks that may be held or waited on concurrently. Provide a hint to systems that use a monitor cache as to how much space to dedicate to the cache.

Signature

```
public static
void setMaximumConcurrentLocks(int numLocks)
```

Parameters

numLocks An integer whose value becomes the number of locks that can be in simultaneous use without incurring an execution time increase. When `number` is less than or equal to zero nothing happens. When the system does not use this hint this method has no effect other than on the value returned by `getMaximumConcurrentLocks()`[16].

[15] Section 13.3.1.6
[16] Section 13.3.1.7.2

setMaximumConcurrentLocks(int, boolean)

Sets the anticipated maximum number of locks that may be held or waited on concurrently. Provide a limit for the size of the monitor cache on systems that provide one when hard is true.

Signature

```
public static
void setMaximumConcurrentLocks(int number, boolean hard)
```

Parameters

number The maximum number of locks that can be in simultaneous use without incurring an execution time increase. When `number` is less than or equal to zero nothing happens. When the system does not use this hint this method has no effect other than on the value returned by `getMaximumConcurrentLocks()`[17].

hard When true, `number` sets a limit. When a lock is attempted which would cause the number of locks to exceed `number` then a `ResourceLimitError`[18] is thrown. When the system does not limit use of concurrent locks, this parameter is silently ignored.

setSecurityManager(RealtimeSecurity)

Sets a new realtime security manager.

Signature

```
public static
void setSecurityManager(RealtimeSecurity manager)
```

Parameters

manager A `RealtimeSecurity`[19] object which will become the new security manager.

Throws

SecurityException when security manager has already been set.

getInitialMonitorControl

[17]Section 13.3.1.7.2
[18]Section 14.2.3.5
[19]Section 13.3.1.6

Returns the monitor control object that represents the initial monitor control policy.
Available since RTSJ 1.0.1

Signature

```
public static
javax.realtime.MonitorControl getInitialMonitorControl()
```

Returns

The initial monitor control policy.

supports(RTSJModule)

Determine if a particular module is supported.
Available since RTSJ 2.0

Signature

```
public static
boolean supports(RTSJModule module)
```

Parameters

module of interest.

Returns

`true` when `module` is supported; otherwise `false`.

13.4 Rationale

This specification accommodates the variation in underlying systems in a number of ways. The `RealtimeSystem` class functions in similar capacity to `java.lang.System`. Similarly, the `RealtimeSecurity` class functions similarly to `java.lang.SecurityManager`. The concept of optionally required classes provides additional flexibility. Such classes provide a commonality that can be relied upon by program logic that intends to execute on implementations that supports a given function, such as `POSIXSignal` and `POSIXRealtimeSignal` encapsulate common functionality for POSIX compliant systems.

Chapter 14

Exceptions

14.1 Overview

This section contains exceptions defined by the `RTSJ`. These exception classes

- provide additional exception classes required for other sections of this specification, and
- provide the ability to asynchronously transfer the control of program logic (see `AsynchronouslyInterruptedException`).

14.1.1 Semantics

This list establishes the semantics that are applicable across the classes of this section. Semantics that apply to particular classes, constructors, methods, and fields will be found in the class description and the constructor, method, and field detail sections.

- All classes in this section are required.
- All exceptions, except `AsynchronouslyInterruptedException`, are required to have semantics exactly as those of their eventual superclass in the `java.*` hierarchy.

The `AsynchronouslyInterruptedException` class is not included in this chapter. It is more closely related to asynchronous operation than to exception handling and so can be found in the Asynchrony chapter.

14.2 Package javax.realtime

14.2.1 Interfaces

14.2.1.1 PreallocatedThrowable

A marker class to indicate that a `Throwable` is intended to be created once and reused. `Throwables` that implement this interface kept their state in a `Schedulable` Object (SO)local data structure instead of the object itself. This means that data is only valid until the next `PreallocatedThrowable` is thrown in the current SO. Having a marker interface makes it easier to provide checking tools to ensure the proper throw sequence for all `Throwables` thrown from user code.

14.2.1.1.1 Methods

---

fillInStackTrace

Calls into the virtual machine to capture the current stack trace in SO local memory.

Signature

```
public
java.lang.Throwable fillInStackTrace()
```

Returns

a reference to this `Throwable`.

getMessage

get the message describing the problem from SO local memory.

Signature

```
public
java.lang.String getMessage()
```

Returns

the message given to the constructor or `null` when no message was set.

getLocalizedMessage

Subclasses may override this message to get an error message that is localized to the default locale.

By default it returns `getMessage()`.

Signature

```
public
java.lang.String getLocalizedMessage()
```

Returns

the value of `getMessage()`.

initMessage(String)

Set the message in SO local storage. This is the only method that is not also defined in `Throwable`.

Signature

```
public
void initMessage(String message)
```

Parameters

message is the text to save.

getCause

getCause returns the cause of this exception or `null` when no cause was set. The cause is another exception that was caught before this exception was created.

Signature

```
public
java.lang.Throwable getCause()
```

Returns

The cause or `null`.

initCause(Throwable)

Initializes the cause to the given Throwable is SO local memory.

Signature

```
public
java.lang.Throwable initCause(Throwable causingThrowable)
```

Parameters

causingThrowable the reason why this Throwable gets Thrown.

Throws

IllegalArgumentException when the cause is this Throwable itself.

Returns

the reference to this Throwable.

printStackTrace

Print stack trace of this Throwable to System.err.

The printed stack trace contains the result of toString() as the first line followed by one line for each stack trace element that contains the name of the method or constructor, optionally followed by the source file name and source file line number when available.

For JamaicaVM, this routine also works before System was initialized by using low-level exception printing mechanisms provided by class com.aicas.jamaica.lang.Debug.

Signature

```
public
void printStackTrace()
```

printStackTrace(PrintStream)

Print the stack trace of this Throwable to the given stream.

The printed stack trace contains the result of toString() as the first line followed by one line for each stack trace element that contains the name of the method or constructor, optionally followed by the source file name and source file line number when available.

For JamaicaVM, when printing to the stream causes another exception, low-level exception printing mechanisms provided by class com.aicas.jamaica.lang.Debug will be used to print the exception to stderr.

Signature

```
public
void printStackTrace(PrintStream stream)
```

Parameters

stream the stream to print to.

printStackTrace(PrintWriter)

Print the stack trace of this Throwable to the given PrintWriter.

The printed stack trace contains the result of toString() as the first line followed by one line for each stack trace element that contains the name of the method or constructor, optionally followed by the source file name and source file line number when available.

For JamaicaVM, when printing to the PrintWriter causes another exception, low-level exception printing mechanisms provided by class com.aicas.jamaica.lang.Debug will be used to print the exception to stderr.

Signature

```
public
void printStackTrace(PrintWriter s)
```

Parameters

s the PrintWriter to write to.

getStackTrace

Get the stack trace created by fillInStackTrace for this Throwable as an array of StackTraceElements.

The stack trace does not need to contain entries for all methods that are actually on the call stack, the virtual machine may decide to skip some stack trace entries. Even an empty array is a valid result of this function.

Repeated calls of this function without intervening calls to fillInStackTrace will return the same result.

For JamaicaVM, the stack trace may omit methods that are compiled by the static compiler. Particularly, methods are compiled and that do not contain an exception handler themselves usually do not require the creation of a stack frame at runtime. To improve performance, no stack frame is generated in these cases.

When memory areas of the RTSJ are used (see `MemoryArea`[1]), and this Throwable was allocated in a different memory area than the current allocation context, the resulting stack trace will be allocated in either the same memory area this was

[1]Section 11.4.3.6

allocated in or the current memory area, depending on which is the least deeply nested, thereby creating objects that are assignment compatible with both areas.

Signature

```
public
java.lang.StackTraceElement[] getStackTrace()
```

Returns

array representing the stack trace, never `null`.

setStackTrace(java.lang.StackTraceElement[])

This method allows overriding the stack trace that was filled during construction of this object. It is intended to be used in a serialization context when the stack trace of a remote exception should be treated like a local.

Signature

```
public
void setStackTrace(java.lang.StackTraceElement[] new_stackTrace)
throws NullPointerException
```

Parameters

new_stackTrace the stack trace to replace be used.

Throws

NullPointerException when new_stackTrace or any element of new_stackTrace is `null`.

14.2.2 Exceptions

14.2.2.1 ArrivalTimeQueueOverflowException

When an arrival time occurs and should be queued but the queue already holds a number of times equal to the initial queue length defined by this an instance of this class may be thrown. When the arrival time is a result of a happening to which the instance of `AsyncEventHandler` is bound then the arrival time is ignored.
Available since RTSJ 1.0.1 Becomes unchecked

Inheritance

java.lang.Object

java.lang.Throwable
  java.lang.Exception
    java.lang.RuntimeException
      javax.realtime.ArrivalTimeQueueOverflowException

14.2.2.1.1 Constructors

ArrivalTimeQueueOverflowException

A constructor for `ArrivalTimeQueueOverflowException`.

Signature

```
public
  ArrivalTimeQueueOverflowException()
```

ArrivalTimeQueueOverflowException(String)

A descriptive constructor for `ArrivalTimeQueueOverflowException`.

Signature

```
public
  ArrivalTimeQueueOverflowException(String description)
```

Parameters

description A description of the exception.

14.2.2.2 CeilingViolationException

This exception is thrown when a schedulable or `java.lang.Thread` attempts to lock an object governed by an instance of `PriorityCeilingEmulation`[2] and the thread or SO's base priority exceeds the policy's ceiling.

[2]Section 7.3.1.2

Inheritance

java.lang.Object
  java.lang.Throwable
    java.lang.Exception
      java.lang.RuntimeException
        java.lang.IllegalArgumentException
          java.lang.IllegalThreadStateException
            javax.realtime.CeilingViolationException

14.2.2.2.1 Constructors

14.2.2.2.2 Methods

getCeiling

Gets the ceiling of the `PriorityCeilingEmulation` policy which was exceeded by the base priority of an SO or thread that attempted to synchronize on an object governed by the policy, which resulted in throwing of `this`.

Signature

```
public
int getCeiling()
```

Returns

The ceiling of the `PriorityCeilingEmulation` policy which caused this exception to be thrown.

getCallerPriority

Gets the base priority of the SO or thread whose attempt to synchronize resulted in the throwing of this.

Signature

```
public
int getCallerPriority()
```

Returns

The synchronizing thread's base priority.

14.2.2.3 DeregistrationException

An exception to throw when trying to deregister an `ActiveEvent`[3] from an `ActiveEventDispatcher`[4] to which it is not registered.

Inheritance

java.lang.Object
  java.lang.Throwable
    java.lang.Exception
      java.lang.RuntimeException
        javax.realtime.DeregistrationException

14.2.2.3.1 Constructors

DeregistrationException

Create an exception with neither message nor cause

Signature

```
public
  DeregistrationException()
```

DeregistrationException(String)

Create an exception with just a message.

Signature

[3] Section 8.4.1.1
[4] Section 8.4.3.3

```
public
  DeregistrationException(String message)
```

Parameters
message a description of the reason for this exception

DeregistrationException(Throwable)

Create an exception with just a cause.

Signature

```
public
  DeregistrationException(Throwable cause)
```

Parameters
cause another exception that caused to this one.

DeregistrationException(String, Throwable)

Create an exception with a message and a cause.

Signature

```
public
  DeregistrationException(String message, Throwable cause)
```

Parameters
message a description of the reason for this exception
cause another exception that caused to this one.

14.2.2.4 DuplicateEventException

Inheritance

java.lang.Object
  java.lang.Throwable
    java.lang.Exception
      javax.realtime.DuplicateEventException

14.2.2.4.1 Constructors

DuplicateEventException

Signature

```
public
  DuplicateEventException()
```

14.2.2.5 DuplicateFilterException

`PhysicalMemoryManager`[5] can only accommodate one filter object for each type of memory. It throws this exception when an attempt is made to register more than one filter for a type of memory.

Inheritance

java.lang.Object
  java.lang.Throwable
    java.lang.Exception
      javax.realtime.DuplicateFilterException

14.2.2.5.1 Constructors

DuplicateFilterException

A constructor for `DuplicateFilterException`.

[5]Section 15.3.2.3

Signature

```
public
  DuplicateFilterException()
```

DuplicateFilterException(String)

A descriptive constructor for `DuplicateFilterException`.

Signature

```
public
  DuplicateFilterException(String description)
```

Parameters

description Description of the error.

14.2.2.6 InaccessibleAreaException

The specified memory area is not on the current thread's scope stack.
Available since RTSJ 1.0.1 Becomes unchecked

Inheritance

java.lang.Object
  java.lang.Throwable
    java.lang.Exception
      java.lang.RuntimeException
        javax.realtime.InaccessibleAreaException

14.2.2.6.1 Constructors

InaccessibleAreaException

A constructor for `InaccessibleAreaException`.

Signature

```
public
  InaccessibleAreaException()
```

InaccessibleAreaException(String)

A descriptive constructor for `InaccessibleAreaException`.

Signature

```
public
  InaccessibleAreaException(String description)
```

Parameters

description Description of the error.

14.2.2.7 LateStartException

Exception thrown when a periodic realtime thread or timer is started after its assigned, absolute, start time.
Available since RTSJ 2.0

Inheritance

java.lang.Object
  java.lang.Throwable
    java.lang.Exception
      javax.realtime.LateStartException

14.2.2.7.1 Constructors

LateStartException

Signature

```
public
  LateStartException()
```

LateStartException(String)

Signature

```
public
  LateStartException(String description)
```

14.2.2.8 MITViolationException

Thrown by the `AsyncEvent.fire()`[6] on a minimum interarrival time violation. More specifically, it is thrown under the semantics of the base priority scheduler's sporadic parameters' `mitViolationExcept` policy when an attempt is made to introduce a release that would violate the MIT constraint.
Available since RTSJ 1.0.1 Becomes unchecked

Inheritance

java.lang.Object
  java.lang.Throwable
    java.lang.Exception
      java.lang.RuntimeException
        javax.realtime.MITViolationException

14.2.2.8.1 Constructors

[6] Section 8.4.3.4.2

MITViolationException

A constructor for `MITViolationException`.

Signature

```
public
  MITViolationException()
```

MITViolationException(String)

A descriptive constructor for `MITViolationException`.

Signature

```
public
  MITViolationException(String description)
```

Parameters

description Description of the error.

14.2.2.9 MemoryInUseException

There has been attempt to allocate a range of physical or virtual memory that is already in use.

Inheritance

java.lang.Object
  java.lang.Throwable
    java.lang.Exception
      java.lang.RuntimeException
        javax.realtime.MemoryInUseException

14.2.2.9.1 Constructors

MemoryInUseException

A constructor for `MemoryInUseException`.

Signature

```
public
  MemoryInUseException()
```

MemoryInUseException(String)

A descriptive constructor for `MemoryInUseException`.

Signature

```
public
  MemoryInUseException(String description)
```

Parameters

description Description of the error.

14.2.2.10 MemoryScopeException

---

when construction of any of the wait-free queues is attempted with the ends of the queue in incompatible memory areas. Also thrown by wait-free queue methods when such an incompatibility is detected after the queue is constructed.

Inheritance

java.lang.Object
  java.lang.Throwable
    java.lang.Exception
      java.lang.RuntimeException
        javax.realtime.MemoryScopeException

14.2.2.10.1 Constructors

MemoryScopeException

A constructor for `MemoryScopeException`.

Signature

```
public
  MemoryScopeException()
```

MemoryScopeException(String)

A descriptive constructor for `MemoryScopeException`.

Signature

```
public
  MemoryScopeException(String description)
```

Parameters

description A description of the exception.

14.2.2.11 MemoryTypeConflictException

This exception is thrown when the `PhysicalMemoryManager`[7] is given conflicting specifications for memory. The conflict can be between types in an array of memory type specifiers, or between the specifiers and a specified base address.
Available since RTSJ 1.0.1 Changed to an unchecked exception.

Inheritance

java.lang.Object
  java.lang.Throwable
    java.lang.Exception
      java.lang.RuntimeException
        javax.realtime.MemoryTypeConflictException

[7]Section 15.3.2.3

14.2.2.11.1 Constructors

MemoryTypeConflictException

A constructor for `MemoryTypeConflictException`.

Signature

```
public
  MemoryTypeConflictException()
```

MemoryTypeConflictException(String)

A descriptive constructor for `MemoryTypeConflictException`.

Signature

```
public
  MemoryTypeConflictException(String description)
```

Parameters

description A description of the exception.

14.2.2.12 OffsetOutOfBoundsException

---

when the constructor of an `ImmortalPhysicalMemory`[8], `LTPhysicalMemory`[9], `VTPhysicalMemory`[10], `RawMemoryAccess`[11], or `RawMemoryFloatAccess`[12] is given an invalid address.

[8] Section 11.4.3.3
[9] Section 11.4.3.5
[10] Section 15.3.2.8
[11] Section 15.3.2.5
[12] Section 15.3.2.6

Available since RTSJ 1.0.1 Becomes unchecked

Inheritance

java.lang.Object
  java.lang.Throwable
    java.lang.Exception
      java.lang.RuntimeException
        javax.realtime.OffsetOutOfBoundsException

14.2.2.12.1 Constructors

OffsetOutOfBoundsException

A constructor for `OffsetOutOfBoundsException`.

Signature

```
public
  OffsetOutOfBoundsException()
```

OffsetOutOfBoundsException(String)

A descriptive constructor for `OffsetOutOfBoundsException`.

Signature

```
public
  OffsetOutOfBoundsException(String description)
```

Parameters

description A description of the exception.

14.2.2.13 ProcessorAffinityException

Exception used to report processor affinity-related errors.
Available since RTSJ 2.0

Inheritance

java.lang.Object
  java.lang.Throwable
    java.lang.Exception
      javax.realtime.ProcessorAffinityException

14.2.2.13.1 Constructors

ProcessorAffinityException

Signature

```
public
  ProcessorAffinityException()
```

ProcessorAffinityException(String)

Signature

```
public
  ProcessorAffinityException(String msg)
```

14.2.2.14 RegistrationException

An exception to throw when trying to register an `ActiveEvent`[13] with an `ActiveEvent-Dispatcher`[14] to which it is already registered.

Inheritance

java.lang.Object
  java.lang.Throwable
    java.lang.Exception
      java.lang.RuntimeException
        javax.realtime.RegistrationException

14.2.2.14.1 Constructors

RegistrationException

Create an exception with neither message nor cause

Signature

```
public
  RegistrationException()
```

RegistrationException(String)

Create an exception with just a message.

Signature

```
public
  RegistrationException(String message)
```

Parameters

message a description of the reason for this exception

[13]Section 8.4.1.1
[14]Section 8.4.3.3

RegistrationException(Throwable)

Create an exception with just a cause.

Signature

```
public
  RegistrationException(Throwable cause)
```

Parameters
cause another exception that caused to this one.

RegistrationException(String, Throwable)

Create an exception with a message and a cause.

Signature

```
public
  RegistrationException(String message, Throwable cause)
```

Parameters
message a description of the reason for this exception
cause another exception that caused to this one.

14.2.2.15 ScopedCycleException

Thrown when a schedulable attempts to enter an instance of `ScopedMemory`[15] where that operation would cause a violation of the single parent rule.

Inheritance
java.lang.Object
  java.lang.Throwable
    java.lang.Exception
      java.lang.RuntimeException
        javax.realtime.ScopedCycleException

[15]Section 11.4.3.11

14.2.2.15.1 Constructors

ScopedCycleException

A constructor for `ScopedCycleException`.

Signature

```
public
  ScopedCycleException()
```

ScopedCycleException(String)

A descriptive constructor for `ScopedCycleException`.

Signature

```
public
  ScopedCycleException(String description)
```

Parameters

description Description of the error.

14.2.2.16 SizeOutOfBoundsException

To throw when the constructor of an `ImmortalPhysicalMemory`[16], `LTPhysicalMemory`[17], or `VTPhysicalMemory`[18] is given an invalid size or when a memory access generated by a raw memory accessor instance (See `RawMemory`[19].) would cause access to an invalid address.

[16] Section 11.4.3.3
[17] Section 11.4.3.5
[18] Section 15.3.2.8
[19] Section 12.4.1.16

Available since RTSJ 1.0.1 Becomes unchecked

Inheritance

java.lang.Object
  java.lang.Throwable
    java.lang.Exception
      java.lang.RuntimeException
        javax.realtime.SizeOutOfBoundsException

14.2.2.16.1 Constructors

SizeOutOfBoundsException

A constructor for `SizeOutOfBoundsException`.

Signature

```
public
  SizeOutOfBoundsException()
```

SizeOutOfBoundsException(String)

A descriptive constructor for `SizeOutOfBoundsException`.

Signature

```
public
  SizeOutOfBoundsException(String description)
```

Parameters

description The description of the exception.

14.2.2.17 UnknownHappeningException

This exception is used to indicate a situation where an instance of `AsyncEvent`[20] attempts to bind to a happening that does not exist.

Inheritance

java.lang.Object
  java.lang.Throwable
    java.lang.Exception
      java.lang.RuntimeException
        javax.realtime.UnknownHappeningException

14.2.2.17.1 Constructors

UnknownHappeningException

A constructor for `UnknownHappeningException`.

Signature

```
public
  UnknownHappeningException()
```

UnknownHappeningException(String)

A descriptive constructor for `UnknownHappeningException`.

Signature

```
public
  UnknownHappeningException(String description)
```

Parameters

description Description of the error.

[20]Section 8.4.3.4

14.2.2.18 UnsupportedPhysicalMemoryException

Thrown when the underlying hardware does not support the type of physical memory requested from an instance of one of the physical memory or raw memory access classes.

See Section RawMemoryAccess

See Section RawMemoryFloatAccess

See Section ImmortalPhysicalMemory

See Section LTPhysicalMemory

See Section VTPhysicalMemory

Available since RTSJ 1.0.1 Becomes unchecked

Inheritance

java.lang.Object
  java.lang.Throwable
    java.lang.Exception
      java.lang.RuntimeException
        javax.realtime.UnsupportedPhysicalMemoryException

14.2.2.18.1 Constructors

UnsupportedPhysicalMemoryException

A constructor for `UnsupportedPhysicalMemoryException`.

Signature

```
public
  UnsupportedPhysicalMemoryException()
```

UnsupportedPhysicalMemoryException(String)

A descriptive constructor for `UnsupportedPhysicalMemoryException`.

Signature

```
public
  UnsupportedPhysicalMemoryException(String description)
```

Parameters

description The description of the exception.

14.2.3 Classes

14.2.3.1 AlignmentError

The exception thrown on an on a request for a raw memory factory to return memory for a base address that is aligned such that the factory cannot guarantee that loads and stores based on that address will meet the factory's specifications. For instance, on many processors, odd addresses are unsuitable for anything but byte access.

Inheritance

java.lang.Object
  java.lang.Throwable
    java.lang.Error
      javax.realtime.AlignmentError

14.2.3.1.1 Constructors

AlignmentError

Signature

```
public
  AlignmentError()
```

14.2.3.2 BacktraceManagement

Provide the static methods for managing the thread local memory used for storing the data needed by preallocated exceptions. Preallocated methods can implement their methods using these methods. User code should not call these methods directly.
Available since RTSJ 2.0

Inheritance
java.lang.Object
javax.realtime.BacktraceManagement

14.2.3.2.1 Constructors

BacktraceManagement

Signature

```
public
  BacktraceManagement()
```

14.2.3.2.2 Methods

fillInStackTrace

Capture the current thread's stack trace and save it in thread local storage. Only the part of the stack trace that fits in the preallocated buffer is stored. This method should be called by a preallocated exception to implement its method of the same name.

Signature

```
public static
void fillInStackTrace()
```

getMessage

Get the message from thread local storage that was saved by the last preallocated exception thrown. This method should be called by a preallocated exception to implement its method of the same name.

Signature

```
public static
java.lang.String getMessage()
```

Returns

the message

initMessage(String)

Save the message in thread local storage for later retrieval. Only the part of the message that fits in the preallocated buffer is stored. This method should be called by a preallocated exception to implement its method of the same name.

Signature

```
public static
void initMessage(String message)
```

Parameters

message the message to save.

getCause

Get the cause from thread local storage that was saved by the last preallocated exception thrown. The actual exception that of the cause is not saved, but just a reference to its type. This returns a newly allocated exception without any valid content, i.e., no valid stack trace. This method should be called by a preallocated exception to implement its method of the same name.

Signature

```
public static
java.lang.Throwable getCause()
```

Returns

the message

initCause(Throwable)

Save the message in thread local storage for later retrieval. Only a reference to the exception class is stored. The rest of its information is lost. This method should be called by a preallocated exception to implement its method of the same name.

Signature

```
public static
void initCause(Throwable causingThrowable)
```

Parameters

causingThrowable

getStackTrace

Get the stack trace from thread local storage that was saved by the last preallocated exception thrown. This method should be called by a preallocated exception to implement its method of the same name.

Signature

```
public static
java.lang.StackTraceElement[] getStackTrace()
```

Returns

an array of the elements of the stack trace.

14.2.3.3 IllegalAssignmentError

The exception thrown on an attempt to make an illegal assignment. For example, this will be thrown on any attempt to assign a reference to an object in scoped memory (an area of memory identified be an instance of `ScopedMemory`[21]) to a field of an object in immortal memory.

Inheritance

java.lang.Object
  java.lang.Throwable
    java.lang.Error
      javax.realtime.IllegalAssignmentError

14.2.3.3.1 Constructors

IllegalAssignmentError

A constructor for `IllegalAssignmentError`.

Signature

```
public
  IllegalAssignmentError()
```

IllegalAssignmentError(String)

A descriptive constructor for `IllegalAssignmentError`.

Signature

```
public
  IllegalAssignmentError(String description)
```

Parameters

description Description of the error.

[21]Section 11.4.3.11

14.2.3.4 MemoryAccessError

This error is thrown on an attempt to refer to an object in an inaccessible `MemoryArea`[22]. For example this will be when logic in a `NoHeapRealtimeThread`[23] attempts to refer to an object in the traditional Java heap.

Inheritance

java.lang.Object
  java.lang.Throwable
    java.lang.Error
      javax.realtime.MemoryAccessError

14.2.3.4.1 Constructors

MemoryAccessError

A constructor for `MemoryAccessError`.

Signature

```
public
  MemoryAccessError()
```

MemoryAccessError(String)

A descriptive constructor for `MemoryAccessError`.

Signature

```
public
  MemoryAccessError(String description)
```

[22]Section 11.4.3.6
[23]Section 15.3.2.1

Parameters

description Description of the error.

14.2.3.5 ResourceLimitError

when an attempt is made to exceed a system resource limit, such as the maximum number of locks.

Inheritance

java.lang.Object
  java.lang.Throwable
    java.lang.Error
      javax.realtime.ResourceLimitError

14.2.3.5.1 Constructors

ResourceLimitError

A constructor for `ResourceLimitError`.

Signature

```
public
  ResourceLimitError()
```

ResourceLimitError(String)

A descriptive constructor for `ResourceLimitError`.

Signature

```
public
  ResourceLimitError(String description)
```

Parameters

description The description of the exception.

14.2.3.6 ThrowBoundaryError

The error thrown by `MemoryArea.enter(Runnable logic)`[24] when a `Throwable` allocated from memory that is not usable in the surrounding scope tries to propagate out of the scope of the `enter`.

Inheritance

java.lang.Object
  java.lang.Throwable
    java.lang.Error
      javax.realtime.ThrowBoundaryError

14.2.3.6.1 Constructors

ThrowBoundaryError

A constructor for `ThrowBoundaryError`.

Signature

```
public
  ThrowBoundaryError()
```

ThrowBoundaryError(String)

A descriptive constructor for `ThrowBoundaryError`.

Signature

[24] Section 11.4.3.6.2

```
public
  ThrowBoundaryError(String description)
```

Parameters

description Description of the error.

14.2.4 Rationale

The need for additional exceptions given the new semantics added by the other sections of this specification is obvious. That the specification attaches new, nonconventional, exception semantics to `AsynchronouslyInterruptedException` is, perhaps, not so obvious. However, after careful thought, and given our self-imposed directive that only well-defined code blocks would be subject to having their control asynchronously transferred, the chosen mechanism is logical.

Chapter 15

Deprecated Classes

15.1 Overview

Since modules are new in 2.0 and this version introduces new ways of handling happening, POSIX signals, and raw memory access, there is no need to include the old API in the `RTSJ` subsets. Therefore the depricated classes have moved here. Only full implementation of the `RTSJ` should implement them.

15.2 Semantics

Implementations of these classes are optional. They are only needed for backward compatibility. They should not be included in implementations that do not include all modules.

15.3 Package javax.realtime

15.3.1 Interfaces

15.3.1.1 PhysicalMemoryName

Tagging interface used to identify objects used to name physical memory types.
Available since RTSJ 2.0

15.3.1.2 PhysicalMemoryTypeFilter

Implementation or device providers may include classes that implement `PhysicalMemoryTypeFilter` which allow additional characteristics of memory in devices to be specified. Implementations of `PhysicalMemoryTypeFilter` are intended to be used by the `PhysicalMemoryManager`[1], not directly from application code.
Deprecated since RTSJ version as of RTSJ 2.0

15.3.1.2.1 Methods

---

contains(long, long)

Queries the system about whether the specified range of memory contains any of this type.
See Section PhysicalMemoryManager.isRemovable

Signature
```
public
boolean contains(long base, long size)
```
Parameters
base The physical address of the beginning of the memory region.

[1]Section 15.3.2.3

size The size of the memory region.

Throws

IllegalArgumentException when `base` or `size` is negative.

OffsetOutOfBoundsException when `base` is less than zero.

SizeOutOfBoundsException when `base` plus `size` would be greater than the physical addressing range of the processor.

Returns

true when the specified range contains ANY of this type of memory.

find(long, long)

Search for physical memory of the right type.

Signature

```
public
long find(long base, long size)
```

Parameters

base The physical address at which to start searching.

size The amount of memory to be found.

Throws

OffsetOutOfBoundsException when `base` is less than zero.

SizeOutOfBoundsException when `base` plus `size` would be greater than the physical addressing range of the processor.

IllegalArgumentException when `base` or `size` is negative.

Returns

The address where memory was found or -1 when it was not found.

getVMAttributes

Gets the virtual memory attributes of `this`. The value of this field is as defined for the POSIX `mmap` function's `prot` parameter for the platform. The meaning of the bits is platform-dependent. POSIX defines constants for PROT_READ, PROT_WRITE, PROT_EXEC, and PROT_NONE.

Signature

```
public
int getVMAttributes()
```

Returns

The virtual memory attributes as an integer.

getVMFlags

Gets the virtual memory flags of `this`. The value of this field is as defined for the POSIX `mmap` function's `flags` parameter for the platform. The meaning of the bits is platform-dependent. POSIX defines constants for MAP_SHARED, MAP_PRIVATE, and MAP_FIXED.

Signature

```
public
int getVMFlags()
```

Returns

The virtual memory flags as an integer.

initialize(long, long, long)

When configuration is required for memory to fit the attribute of this object, do the configuration here.

Signature

```
public
void initialize(long base, long vBase, long size)
```

Parameters

base The address of the beginning of the physical memory region.
vBase The address of the beginning of the virtual memory region.
size The size of the memory region.

Throws

IllegalArgumentException when `base` or `size` is negative.
OffsetOutOfBoundsException when `base` is less than zero.
SizeOutOfBoundsException when `base` plus `size` would be greater than the physical addressing range of the processor, or `vBase` plus `size` would exceed the virtual addressing range of the processor.

isPresent(long, long)

Queries the system about the existence of the specified range of physical memory.

See Section PhysicalMemoryManager.isRemoved

Signature

`public`
`boolean isPresent(long base, long size)`

Parameters

base The address of the beginning of the memory region.
size The size of the memory region.

Throws

IllegalArgumentException when the base and size do not fall into this type of memory.
OffsetOutOfBoundsException when `base` is less than zero.
SizeOutOfBoundsException when `base` plus `size` would be greater than the physical addressing range of the processor.

Returns

True when all of the memory is present. False when any of the memory has been removed.

isRemovable

Queries the system about the removability of this memory.

Signature

`public`
`boolean isRemovable()`

Returns

true when this type of memory is removable.

onInsertion(long, long, AsyncEvent)

Register the specified `AsyncEvent`[2] to fire when any memory of this type in the range is added to the system.
Available since RTSJ 1.0.1

[2]Section 8.4.3.4

Signature

```
public
void onInsertion(long base, long size, AsyncEvent ae)
```

Parameters

base The starting address in physical memory.
size The size of the memory area.
ae The async event to fire.

Throws

IllegalArgumentException when `ae` is `null`, or when the specified range contains no removable memory of this type. `IllegalArgumentException` may also be thrown when `size` is less than zero.
OffsetOutOfBoundsException when `base` is less than zero.
SizeOutOfBoundsException when `base` plus `size` would be greater than the physical addressing range of the processor.

onRemoval(long, long, AsyncEvent)

Register the specified AE to fire when any memory in the range is removed from the system.
Available since RTSJ 1.0.1

Signature

```
public
void onRemoval(long base, long size, AsyncEvent ae)
```

Parameters

base The starting address in physical memory.
size The size of the memory area.
ae The async event to register.

Throws

IllegalArgumentException when the specified range contains no removable memory of this type, when `ae` is `null`, or when `size` is less than zero.
OffsetOutOfBoundsException when `base` is less than zero.
SizeOutOfBoundsException when `base` plus `size` would be greater than the physical addressing range of the processor.

unregisterInsertionEvent(long, long, AsyncEvent)

Unregister the specified insertion event. The event is only unregistered when all three arguments match the arguments used to register the event, except that `ae` of `null` matches all values of `ae` and will unregister every `ae` that matches the address range.

Note: This method has no effect on handlers registered directly as async event handlers.

Available since RTSJ 1.0.1

Signature

```
public
boolean unregisterInsertionEvent(long base, long size,
AsyncEvent ae)
```

Parameters

base The starting address in physical memory associated with `ae`.

size The size of the memory area associated with `ae`.

ae The event to unregister.

Throws

IllegalArgumentException when `size` is less than 0.

OffsetOutOfBoundsException when `base` is less than zero.

SizeOutOfBoundsException when `base` plus `size` would be greater than the physical addressing range of the processor.

Returns

True when at least one event matched the pattern, false when no such event was found.

unregisterRemovalEvent(long, long, AsyncEvent)

Unregister the specified removal event. The async event is only unregistered when all three arguments match the arguments used to register the event, except that `ae` of `null` matches all values of `ae` and will unregister every `ae` that matches the address range. Note: This method has no effect on handlers registered directly as async event handlers.

Available since RTSJ 1.0.1

Signature

```
public
```

```
boolean unregisterRemovalEvent(long base, long size, AsyncEvent
ae)
```

Parameters

base The starting address in physical memory associated with `ae`.
size The size of the memory area associated with `ae`.
ae The async event to unregister.

Throws

IllegalArgumentException when `size` is less than 0.
OffsetOutOfBoundsException when `base` is less than zero.
SizeOutOfBoundsException when `base` plus `size` would be greater than the physical addressing range of the processor.

Returns

True when at least one event matched the pattern, false when no such event was found.

vFind(long, long)

Search for virtual memory of the right type. This is important for systems where attributes are associated with particular ranges of virtual memory.

Signature

```
public
long vFind(long base, long size)
```

Parameters

base The address at which to start searching.
size The amount of memory to be found.

Throws

OffsetOutOfBoundsException when `base` is less than zero.
SizeOutOfBoundsException when `base` plus `size` would be greater than the physical addressing range of the processor.
IllegalArgumentException when `base` or `size` is negative. `IllegalArgumentException` may also be when `base` is an invalid virtual address.

Returns

The address where memory was found or -1 when it was not found.

15.3.2 Classes

15.3.2.1 NoHeapRealtimeThread

A `NoHeapRealtimeThread` is a specialized form of `RealtimeThread`[3]. Because an instance of `NoHeapRealtimeThread` may immediately preempt any implemented garbage collector, logic contained in its `run()` is never allowed to allocate or reference any object allocated in the heap. At the byte-code level, it is illegal for a reference to an object allocated in heap to appear on a no-heap realtime thread's operand stack.

Thus, it is always safe for a `NoHeapRealtimeThread` to interrupt the garbage collector at any time, without waiting for the end of the garbage collection cycle or a defined preemption point. Due to these restrictions, a `NoHeapRealtimeThread` object must be placed in a memory area such that thread logic may unexceptionally access instance variables and such that Java methods on `Thread` (e.g., enumerate and join) complete normally except where execution would cause access violations. The constructors of `NoHeapRealtimeThread` require a reference to `ScopedMemory`[4] or `ImmortalMemory`[5].

When the thread is started, all execution occurs in the scope of the given memory area. Thus, all memory allocation performed with the *new* operator is taken from this given area.

Deprecated since RTSJ version since RTSJ 2.0

Inheritance

java.lang.Object
  java.lang.Thread
    javax.realtime.RealtimeThread
      javax.realtime.NoHeapRealtimeThread

15.3.2.1.1 Constructors

NoHeapRealtimeThread(SchedulingParameters, ReleaseParameters, MemoryParameters, MemoryArea, ProcessingGroupParameters, Runnable)

[3]Section 5.3.2.2
[4]Section 11.4.3.11
[5]Section 11.4.3.2

Create a realtime thread with the given characteristics and a Runnable. The thread group of the new thread is (effectively) null. The newly-created no-heap real-time thread is associated with the scheduler in effect during execution of the constructor.

Signature

```
public
  NoHeapRealtimeThread(SchedulingParameters scheduling, ReleaseParameters release, M
```

Parameters

scheduling the SchedulingParameters associated with this (and possibly other instances of Schedulable). When scheduling is null, the default is a copy of the creator's scheduling parameters created in the same memory area as the new NoHeapRealtimeThread.

release the ReleaseParameters associated with this (and possibly other instances of Schedulable). When release is null the it defaults to the a copy of the creator's release parameters created in the same memory area as the new NoHeapRealtimeThread.

memory the MemoryParameters associated with this (and possibly other instances of Schedulable). When memory is null, the new NoHeapRealtimeThread will have a null value for its memory parameters, and the amount or rate of memory allocation is unrestricted.

area the MemoryArea associated with this. When area is null, an IllegalArgumentException is thrown.

group the ProcessingGroupParameters associated with this (and possibly other instances of Schedulable). When null, the new NoHeapRealtimeThread will not be associated with any processing group.

logic the Runnable object whose run() method will serve as the logic for the new NoHeapRealtimeThread. When logic is null, the run() method in the new object will serve as its logic.

Throws

IllegalArgumentException when the parameters are not compatible with the associated scheduler, when area is null, when area is heap memory, when area, scheduling release, memory or group is allocated in heap memory. when this is in heap memory, or when logic is in heap memory.

IllegalAssignmentError when the new NoHeapRealtimeThread instance cannot hold references to non-null values of the scheduling release, memory and group, or when those parameters cannot hold a reference to the new NoHeapRealtimeThread.

NoHeapRealtimeThread(SchedulingParameters, ReleaseParameters, MemoryArea)

Create a no-heap realtime thread with the given `SchedulingParameters`[6], `ReleaseParameters`[7] and `MemoryArea`[8], and default values for all other parameters. This constructor is equivalent to `NoHeapRealtimeThread(scheduling, release, null, area, null, null, null)`.

Signature

```
public
  NoHeapRealtimeThread(SchedulingParameters scheduling, ReleaseParameters relea
```

NoHeapRealtimeThread(SchedulingParameters, MemoryArea)

Create a realtime thread with the given `SchedulingParameters`[9] and `MemoryArea`[10] and default values for all other parameters.

This constructor is equivalent to `NoHeapRealtimeThread(scheduling, null, null, area, null, null, null)`.

Signature

```
public
  NoHeapRealtimeThread(SchedulingParameters scheduling, MemoryArea area)
```

15.3.2.1.2 Methods

---

[6]Section 6.4.2.10
[7]Section 6.4.2.8
[8]Section 11.4.3.6
[9]Section 6.4.2.10
[10]Section 11.4.3.6

start

See Section RealtimeThread.start()

Signature

```
public
void start()
```

startPeriodic(PhasingPolicy)

See Section RealtimeThread.startPeriodic(PhasingPolicy)

Available since RTSJ 2.0

Signature

```
public
void startPeriodic(PhasingPolicy phasingPolicy)
throws LateStartException
```

15.3.2.2 POSIXSignalHandler

Jamaica Real-Time Specification for Java class POSIXSignalHandler.

This class permits the use of an AsyncEventHandler to react on the occurrence of POSIX signals.

On systems that support POSIX signals fully, the 13 signals required by POSIX will be supported. Any further signals defined in this class may be supported by the system. On systems that do not support POSIX signals, even the 13 standard signals may never be fired.

Inheritance

java.lang.Object
  javax.realtime.POSIXSignalHandler

15.3.2.2.1 Fields

SIGHUP

`public static final SIGHUP`
Hangup (POSIX).

SIGINT

`public static final SIGINT`
interrupt (ANSI)

SIGQUIT

`public static final SIGQUIT`
quit (POSIX)

SIGILL

`public static final SIGILL`
illegal instruction (ANSI)

SIGTRAP

`public static final SIGTRAP`
trace trap (POSIX), optional signal.

SIGABRT

`public static final SIGABRT`
Abort (ANSI).

SIGBUS

`public static final SIGBUS`
BUS error (4.2 BSD), optional signal.

SIGFPE

```
public static final SIGFPE
```

floating point exception

SIGKILL

```
public static final SIGKILL
```

Kill, unblockable (POSIX).

SIGUSR1

```
public static final SIGUSR1
```

User-defined signal 1 (POSIX).

SIGSEGV

```
public static final SIGSEGV
```

Segmentation violation (ANSI).

SIGUSR2

```
public static final SIGUSR2
```

User-defined signal 2 (POSIX).

SIGPIPE

```
public static final SIGPIPE
```

Broken pipe (POSIX).

SIGALRM

```
public static final SIGALRM
```

Alarm clock (POSIX).

SIGTERM

```
public static final SIGTERM
```

Termination (ANSI).

SIGCHLD

`public static final SIGCHLD`
Child status has changed (POSIX).

SIGCONT

`public static final SIGCONT`
Continue (POSIX), optional signal.

SIGSTOP

`public static final SIGSTOP`
Stop, unblockable (POSIX), optional signal.

SIGTSTP

`public static final SIGTSTP`
Keyboard stop (POSIX), optional signal.

SIGTTIN

`public static final SIGTTIN`
Background read from tty (POSIX), optional signal.

SIGTTOU

`public static final SIGTTOU`
Background write to tty (POSIX), optional signal.

SIGURG

`public static final SIGURG`
Urgent condition on socket (4.2 BSD).
Deprecated since RTSJ version as of RTSJ 1.0.1 not part of POSIX 9945-1-1996 standard

SIGXCPU

`public static final SIGXCPU`

CPU limit exceeded (4.2 BSD).

Deprecated since RTSJ version as of RTSJ 1.0.1 not part of POSIX 9945-1-1996 standard

SIGXFSZ

`public static final SIGXFSZ`

File size limit exceeded (4.2 BSD).

Deprecated since RTSJ version as of RTSJ 1.0.1 not part of POSIX 9945-1-1996 standard

SIGVTALRM

`public static final SIGVTALRM`

Virtual alarm clock (4.2 BSD).

Deprecated since RTSJ version as of RTSJ 1.0.1 not part of POSIX 9945-1-1996 standard

SIGPROF

`public static final SIGPROF`

Profiling alarm clock (4.2 BSD).

Deprecated since RTSJ version as of RTSJ 1.0.1 not part of POSIX 9945-1-1996 standard

SIGWINCH

`public static final SIGWINCH`

Window size change (4.3 BSD, Sun).

Deprecated since RTSJ version as of RTSJ 1.0.1 not part of POSIX 9945-1-1996 standard

SIGIO

`public static final SIGIO`

I/O now possible (4.2 BSD).

Deprecated since RTSJ version as of RTSJ 1.0.1 not part of POSIX 9945-1-1996 standard

SIGPWR

`public static final SIGPWR`

Power failure restart (System V).

Deprecated since RTSJ version as of RTSJ 1.0.1 not part of POSIX 9945-1-1996 standard

SIGSYS

`public static final SIGSYS`

Bad system call, optional signal.

SIGIOT

`public static final SIGIOT`

IOT instruction (4.2 BSD), optional signal.

SIGPOLL

`public static final SIGPOLL`

Pollable event occurred (System V).

Deprecated since RTSJ version as of RTSJ 1.0.1 not part of POSIX 9945-1-1996 standard

SIGCLD

`public static final SIGCLD`

Same as SIGCHLD (System V), optional signal.

SIGEMT

```
public static final SIGEMT
```

EMT instruction, optional signal.

SIGLOST

```
public static final SIGLOST
```

Deprecated since RTSJ version as of RTSJ 1.0.1 not part of POSIX 9945-1-1996 standard

SIGCANCEL

```
public static final SIGCANCEL
```

Deprecated since RTSJ version as of RTSJ 1.0.1 not part of POSIX 9945-1-1996 standard

SIGFREEZE

```
public static final SIGFREEZE
```

Deprecated since RTSJ version as of RTSJ 1.0.1 not part of POSIX 9945-1-1996 standard

SIGLWP

```
public static final SIGLWP
```

Deprecated since RTSJ version as of RTSJ 1.0.1 not part of POSIX 9945-1-1996 standard

SIGTHAW

```
public static final SIGTHAW
```

Deprecated since RTSJ version as of RTSJ 1.0.1 not part of POSIX 9945-1-1996 standard

SIGWAITING

`public static final SIGWAITING`

Deprecated since RTSJ version as of RTSJ 1.0.1 not part of POSIX 9945-1-1996 standard

15.3.2.2.2 Constructors

POSIXSignalHandler

Signature

```
public
  POSIXSignalHandler()
```

15.3.2.2.3 Methods

addHandler(int, AsyncEventHandler)

addHandler adds the handler provided to the set of handlers that will be released on the provided signal.

Signature

```
public static
void addHandler(int signal, AsyncEventHandler handler)
```

Parameters

signal The POSIX signal as defined in the constants SIG*.
handler the handler to be released on the given signal.

Throws

IllegalArgumentException iff signal is not defined by any of the constants in this class or handler is `null`.

removeHandler(int, AsyncEventHandler)

removeHandler removes a handler that was added for a given signal.

Signature

```
public static
void removeHandler(int signal, AsyncEventHandler handler)
```

Parameters

signal The POSIX signal as defined in the constants SIG*.

handler the handler to be removed from the given signal. When this handler is `null` or has not been added to the signal, nothing will happen.

Throws

IllegalArgumentException iff signal is not defined by any of the constants in this class.

setHandler(int, AsyncEventHandler)

setHandler sets the set of handlers that will be released on the provided signal to the set with the provided handler being the single element.

Signature

```
public static
void setHandler(int signal, AsyncEventHandler handler)
```

Parameters

signal The POSIX signal as defined in the constants SIG*.

handler the handler to be released on the given signal, `null` to remove all handlers for the given signal.

Throws

IllegalArgumentException iff signal is not defined by any of the constants in this class.

15.3.2.3 PhysicalMemoryManager

The `PhysicalMemoryManager` is not ordinarily used by applications, except that the implementation may require the application to use the registerFilter[11] method to make the physical memory manager aware of the memory types on their platform. The PhysicalMemoryManager class is primarily intended for use by the various physical memory accessor objects (VTPhysicalMemory[12], LTPhysicalMemory[13], and ImmortalPhysicalMemory[14]) to create objects of the types requested by the application. The physical memory manager is responsible for finding areas of physical memory with the appropriate characteristics and access rights, and moderating any required combination of physical and virtual memory characteristics.

The Physical Memory Manager assumes that the physical adresss space is linear but not necessarily contiguous. That is, addresses range from 0 .. MAX_LONG but there may be holes in the memory space. Some of these holes may be filled with removable memory.

The physical memory is partitioned into chunks (pages, segments, etc.). Each chunk of memory has a base address and a length.

Each chunk of memory has certain properties. Some of these properties may require actions to be performed by the Physical Memory Manager when the memory is accessed. For example, access to IO_PAGE may require the use of special instructions to even reach the devices, or it may require special code sequences to ensure proper handling of processor write queues and caches.

Filters tell the Physical Memory Manager about the properties of the memory that are available on the machine by registering with the Physical Memory Manager.

When the program requests a physical memory area with particular properties, the constructor communicates with the Physical Memory Manager through a private interface. The Physical Memory Manager asks the filter if the the address specified has the required properties and whether it is free, or asks for a chunk of memory with the requested size.

The Physical Memory Manager then maps the physical memory chunk into virtual memory (on systems that support virtual memory). and locks the virtual memory to the memory chunk.

Examples of characteristics that might be specified are: DMA memory, hardware byte swapping, non-cached access to memory, etc. Standard "names" for some memory characteristics are included in this class — DMA, SHARED, ALIGNED, BYTESWAP, and IO_PAGE — support for these characteristics is optional, but when they are supported they must use these names. Additional characteristics may be supported, but only names defined in this specification may be visible in the PhysicalMemoryManager API.

The base implementation will provide a PhysicalMemoryManager.

[11]Section 15.3.2.3.2
[12]Section 15.3.2.8
[13]Section 11.4.3.5
[14]Section 11.4.3.3

Original Equipment Manufacturers or other interested parties may provide `PhysicalMemoryTypeFilter`[15] classes that allow additional characteristics of memory devices to be specified.
Deprecated since RTSJ version as of RTSJ 2.0

Inheritance

java.lang.Object
  javax.realtime.PhysicalMemoryManager

15.3.2.3.1 Fields

ALIGNED

`public static final ALIGNED`

When aligned memory is supported by the implementation specify `ALIGNED` to identify aligned memory. This type of memory ignores low-order bits in load and store accesses to force accesses to fall on natural boundaries for the access type even when the processor uses a poorly aligned address.
Deprecated since RTSJ version as of RTSJ 2.0 This is only applicable to raw memory. Use `RawMemory`[16].

BYTESWAP

`public static final BYTESWAP`

When automatic byte swapping is supported by the implementation specify `BYTESWAP` when byte swapping should be used. Byte-swapping memory re-orders the bytes in accesses for 16 bits or more such that little-endian data in memory is accessed as big-endian, and vice-versa. Such memory would typically be available in swapped mode in one physical address range and in un-swapped mode in another address range.
Deprecated since RTSJ version as of RTSJ 2.0 This is only applicable to raw memory. Use `RawMemory`[17].

[15] Section 15.3.1.2
[16] Section 12.4.1.16
[17] Section 12.4.1.16

DMA

`public static final DMA`

When DMA (Direct Memory Access) memory is supported by the implementation, specify `DMA` to identify DMA memory. This memory is visible to devices that use DMA. In some systems, only a portion of the physical address space is available to DMA devices. On such systems, memory that will be used for DMA must be allocated from the range of addresses that DMA can reach.
Deprecated since RTSJ version as of RTSJ 2.0 This is only applicable to raw memory. Use `RawMemory`[18].

IO_PAGE

`public static final IO_PAGE`

When access to the system I/O space is supported by the implementation specify `IO_PAGE` when I/O space should be used. Addresses tagged with the name `IO_PAGE` are used for memory mapped I/O devices. Such addresses are almost certainly not suitable for physical memory, but only for raw memory access.
Available since RTSJ 1.0.1

SHARED

`public static final SHARED`

When shared memory is supported by the implementation specify `SHARED` to identify shared memory. In a NUMA (Non-Uniform Memory Access) architecture, processors may make some part of their local memory available to other processors. This memory would be tagged with `SHARED`, as would memory that is shared and non-local.

A fully built-out NUMA system might well need sub-classifications of `SHARED` to reflect different paths to memory. Note that, as with other physical memory names, a single byte of memory may be visible at several physical addresses with different access properties at each address. For instance, a byte of shared memory accesses at address x might be shared with high-performance access, but without the support of coherent caches. The same byte accessed at address y might be shared with coherent cache support, but substantially longer access times.

[18] Section 12.4.1.16

15.3.2.3.2 Methods

isRemovable(long, long)

Queries the system about the removability of the specified range of memory.

Signature

```
public static
boolean isRemovable(long base, long size)
```

Parameters

base The starting address in physical memory.
size The size of the memory area.

Throws

IllegalArgumentException when `size` is less than zero.
SizeOutOfBoundsException when `base` plus `size` would be greater than the physical addressing range of the processor.
OffsetOutOfBoundsException when `base` is less than zero.

Returns

true when any part of the specified range can be removed.

isRemoved(long, long)

Queries the system about the removed state of the specified range of memory. This method is used for devices that lie in the memory address space and can be removed while the system is running. (Such as PC cards).

Signature

```
public static
boolean isRemoved(long base, long size)
```

Parameters

base The starting address in physical memory.
size The size of the memory area.

Throws

IllegalArgumentException when `size` is less than zero.
OffsetOutOfBoundsException when `base` is less than zero.

SizeOutOfBoundsException when `base` plus `size` would be greater than the physical addressing range of the processor.

Returns

true when any part of the specified range is currently not usable.

onInsertion(long, long, AsyncEvent)

Register the specified AsyncEvent[19] to fire when any memory in the range is added to the system. When the specified range of physical memory contains multiple different types of removable memory, the AE will be registered with each of them.
Available since RTSJ 1.0.1

Signature

```
public static
void onInsertion(long base, long size, AsyncEvent ae)
```

Parameters

base The starting address in physical memory.

size The size of the memory area.

ae The async event to fire.

Throws

IllegalArgumentException when `ae` is `null`, or when the specified range contains no removable memory, or when `size` is less than zero.

OffsetOutOfBoundsException when `base` is less than zero.

SizeOutOfBoundsException when `base` plus `size` would be greater than the physical addressing range of the processor.

onInsertion(long, long, AsyncEventHandler)

Register the specified AsyncEventHandler[20] to run when any memory in the range is added to the system. When the specified range of physical memory contains multiple different types of removable memory, the AEH will be registered with each of them. When the size or the base is less than 0, unregister all "onInsertion" references to the handler.

Note: This method only removes handlers that were registered with the same method. It has no effect on handlers that were registered using an associated async event.

[19]Section 8.4.3.4

[20]Section 8.4.3.5

Signature

```
public static
void onInsertion(long base, long size, AsyncEventHandler aeh)
```

Parameters

base The starting address in physical memory.
size The size of the memory area.
aeh The handler to register.

Throws

IllegalArgumentException when `aeh` is `null`, or when the specified range contains no removable memory, or when `aeh` is `null` and `size` and `base` are both greater than or equal to zero.
SizeOutOfBoundsException when `base` plus `size` would be greater than the physical addressing range of the processor.

onRemoval(long, long, AsyncEvent)

Register the specified AE to fire when any memory in the range is removed from the system. When the specified range of physical memory contains multiple different types of removable memory, the AE will be registered with each of them.

Signature

```
public static
void onRemoval(long base, long size, AsyncEvent ae)
```

Parameters

base The starting address in physical memory.
size The size of the memory area.
ae The async event to register.

Throws

IllegalArgumentException when the specified range contains no removable memory, when `ae` is `null`, or when `size` is less than zero.
OffsetOutOfBoundsException when `base` is less than zero.
SizeOutOfBoundsException when `base` plus `size` would be greater than the physical addressing range of the processor.

onRemoval(long, long, AsyncEventHandler)

Register the specified AEH to run when any memory in the range is removed from the system. When the specified range of physical memory contains multiple different types of removable memory, the AEH will be registered with each of them. When `size` or `base` is less than 0, unregister all "onRemoval" references to the handler parameter.

Note: This method only removes handlers that were registered with the same method. It has no effect on handlers that were registered using an associated async event.

Signature

```
public static
void onRemoval(long base, long size, AsyncEventHandler aeh)
```

Parameters

base The starting address in physical memory.
size The size of the memory area.
aeh The handler to register.

Throws

IllegalArgumentException when the specified range contains no removable memory, or when `aeh` is `null` and `size` and `base` are both greater than or equal to zero.
SizeOutOfBoundsException when `base` plus `size` would be greater than the physical addressing range of the processor.

registerFilter(Object, PhysicalMemoryTypeFilter)

Register a memory type filter with the physical memory manager.

Values of `name` are compared using reference equality (==) not value equality (`equals()`).

Signature

```
public static final
void registerFilter(Object name, PhysicalMemoryTypeFilter
filter)
throws DuplicateFilterException
```

Parameters

name The type of memory handled by this filter.
filter The filter object.

Throws

DuplicateFilterException when a filter for this type of memory already exists.

ResourceLimitError when the system is configured for a bounded number of filters. This filter exceeds the bound.
IllegalArgumentException when the name parameter is an array of objects, when the name and filter are not both in immortal memory, or when either `name` or `filter` is `null`.
SecurityException when this operation is not permitted.

removeFilter(Object)

Remove the identified filter from the set of registered filters. When the filter is not registered, silently do nothing.

Values of `name` are compared using reference equality (==) not value equality (`equals()`).

Signature

```
public static final
void removeFilter(Object name)
```

Parameters

name The identifying object for this memory attribute.

Throws

IllegalArgumentException when `name` is `null`.
SecurityException when this operation is not permitted.

unregisterInsertionEvent(long, long, AsyncEvent)

Unregister the specified insertion event. The event is only unregistered when all three arguments match the arguments used to register the event, except that `ae` of `null` matches all values of `ae` and will unregister every `ae` that matches the address range.

Note: This method has no effect on handlers registered directly as async event handlers.

Available since RTSJ 1.0.1

Signature

```
public static
boolean unregisterInsertionEvent(long base, long size,
AsyncEvent ae)
```

Parameters

base The starting address in physical memory associated with `ae`.
size The size of the memory area associated with `ae`.
ae The event to unregister.

Throws

IllegalArgumentException when `size` is less than 0.
OffsetOutOfBoundsException when `base` is less than zero.
SizeOutOfBoundsException when `base` plus `size` would be greater than the physical addressing range of the processor.

Returns

True when at least one event matched the pattern, false when no such event was found.

unregisterRemovalEvent(long, long, AsyncEvent)

Unregister the specified removal event. The async event is only unregistered when all three arguments match the arguments used to register the event, except that `ae` of `null` matches all values of `ae` and will unregister every `ae` that matches the address range.

Note: This method has no effect on handlers registered directly as async event handlers.

Available since RTSJ 1.0.1

Signature

```
public static
boolean unregisterRemovalEvent(long base, long size, AsyncEvent
ae)
```

Parameters

base The starting address in physical memory associated with `ae`.
size The size of the memory area associated with `ae`.
ae The async event to unregister.

Throws

IllegalArgumentException when `size` is less than 0.
OffsetOutOfBoundsException when `base` is less than zero.
SizeOutOfBoundsException when `base` plus `size` would be greater than the physical addressing range of the processor.

Returns

True when at least one event matched the pattern, false when no such event was found.

15.3.2.4 RationalTime

An object that represents a time interval milliseconds/10^3 + nanoseconds/10^9 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.

Caution: This class is explicitly unsafe for multithreading when being changed. Code that mutates instances of this class should synchronize at a higher level. **Deprecated since RTSJ version as of RTSJ 1.0.1**

Inheritance

java.lang.Object
  javax.realtime.HighResolutionTime
    javax.realtime.RelativeTime
      javax.realtime.RationalTime

15.3.2.5 RawMemoryAccess

An instance of `RawMemoryAccess` models a range of physical memory as a fixed sequence of bytes. A complement of accessor methods enable the contents of the physical area to be accessed through offsets from the base, interpreted as byte, short, int, or long data values or as arrays of these types.

Whether an offset addresses the high-order or low-order byte is normally based on the value of the `RealtimeSystem.BYTE_ORDER`[21] static byte variable in class `RealtimeSystem`[22]. When the type of memory used for this `RawMemoryAccess` region implements non-standard byte ordering, accessor methods in this class continue to select bytes starting at `offset` from the base address and continuing toward greater addresses. The memory type may control the mapping of these bytes into the primitive data type. The memory type could even select bytes that are not contiguous. In

[21]Section 13.3.1.7.1
[22]Section 13.3.1.7

each case the documentation for the `PhysicalMemoryTypeFilter`[23] must document any mapping other than the "normal" one specified above.

The `RawMemoryAccess` class allows a realtime program to implement device drivers, memory-mapped I/O, flash memory, battery-backed RAM, and similar low-level software.

A raw memory area cannot contain references to Java objects. Such a capability would be unsafe (since it could be used to defeat Java's type checking) and error-prone (since it is sensitive to the specific representational choices made by the Java compiler).

Many of the constructors and methods in this class throw `OffsetOutOfBoundsException`[24]. This exception means that the value given in the offset parameter is either negative or outside the memory area.

Many of the constructors and methods in this class throw `SizeOutOfBoundsException`[25]. This exception means that the value given in the size parameter is either negative, larger than an allowable range, or would cause an accessor method to access an address outside of the memory area.

Unlike other integral parameters in this chapter, negative values are valid for `byte, short, int,` and `long` values that are copied in and out of memory by the `set` and `get` methods of this class.

All offset values used in this class are measured in bytes.

Atomic loads and stores on raw memory are defined in terms of physical memory. This memory may be accessible to threads outside the JVM and to non-programmed access (e.g., DMA), consequently atomic access must be supported by hardware. This specification is written with the assumption that all suitable hardware platforms support atomic loads for aligned bytes, shorts, and ints. Atomic access beyond the specified minimum may be supported by the implementation.

Storing values into raw memory is more hardware-dependent than loading values. Many processor architectures do not support atomic stores of variables except for aligned stores of the processor's word size. For instance, storing a byte into memory might require reading a 32-bit quantity into a processor register, updating the register to reflect the new byte value, then re-storing the whole 32-bit quantity. Changes to other bytes in the 32-bit quantity that take place between the load and the store will be lost.

Some processors have mechanisms that can be used to implement an atomic store of a byte, but those mechanisms are often slow and not universally supported.

This class supports unaligned access to data, but it does not require the implementation to make such access atomic. Accesses to data aligned on its natural

[23] Section 15.3.1.2
[24] Section 14.2.2.12
[25] Section 14.2.2.16

boundary will be atomic when the processor implements atomic loads and stores of that data size.

Except where noted, accesses to raw memory are not atomic with respect to the memory or with respect to schedulables. A raw memory area could be updated by another schedulable, or even unmapped in the middle of a method.

The characteristics of raw-memory access are necessarily platform dependent. This specification provides a minimum requirement for the RTSJ platform, but it also supports optional system properties that identify a platform's level of support for atomic raw put and get. The properties represent a four-dimensional sparse array with boolean values indicating whether that combination of access attributes is atomic. The default value for array entries is false. The dimension are

Attribute	Values	Comment
Access type	read, write	
Data type	• byte, • short, • int, • long, • float, • double	
Alignment	0 to 7	For each data type, the possible alignments range from • 0 == aligned • to data size - 1 == only the first byte of the data is *alignment* bytes away from natural alignment.
Atomicity	• processor, • smp, • memory	• *processor* means access is atomic with respect to other schedulables on that processor. • *smp* means that access is *processor* atomic, and atomic with respect across the processors in an SMP. • *memory* means that access is *smp* atomic, and atomic with respect to all access to the memory including DMA.

The true values in the table are represented by properties of the following form. javax.realtime.atomicaccess_<access>_<type>_<alignment>_atomicity=true for example:

javax.realtime.atomicaccess_read_byte_0_memory=true

Table entries with a value of false may be explicitly represented, but since false is the default value, such properties are redundant.

All raw memory access is treated as volatile, and *serialized*. The run-time must be forced to re-read memory or write to memory on each call to a raw memory getxxx or putxxx method, and to complete the reads and writes in the order they appear in the program order.
Deprecated since RTSJ version as of RTSJ 2.0. Use `RawMemoryFactory`[26] to create the appropriate `RawMemory`[27] object.

Inheritance

java.lang.Object
  javax.realtime.RawMemoryAccess

15.3.2.5.1 Constructors

RawMemoryAccess(Object, long)

Construct an instance of `RawMemoryAccess` with the given parameters, and set the object to the mapped state. When the platform supports virtual memory, map the raw memory into virtual memory.

The run time environment is allowed to choose the virtual address where the raw memory area corresponding to this object will be mapped. The attributes of the mapping operation are controlled by the vMFlags and vMAttributes of the `PhysicalMemoryTypeFilter` objects that matched this object's `type` parameter. (See `PhysicalMemoryTypeFilter.getVMAttributes`[28] and `PhysicalMemoryType-Filter.getVMFlags`[29].
Deprecated since RTSJ version as of RTSJ 2.0. Use `RawMemoryFactory`[30] to create the appropriate `RawMemory`[31] object.

Signature

```
public
  RawMemoryAccess(Object type, long size)
```

[26]Section 12.4.3.5
[27]Section 12.4.1.16
[28]Section 15.3.1.2.1
[29]Section 15.3.1.2.1
[30]Section 12.4.3.5
[31]Section 12.4.1.16

Parameters

type An instance of `Object` representing the type of memory required (e.g., *dma, shared*) - used to define the base address and control the mapping. When the required memory has more than one attribute, `type` may be an array of objects. When `type` is `null` or a reference to an array with no entries, any type of memory is acceptable. Note that `type` values are compared by reference (==), not by value (`equals`).

size The size of the area in bytes.

Throws

SecurityException when the application doesn't have permissions to access physical memory, the specified range of addresses, or the given type of memory.

SizeOutOfBoundsException when the size is negative or extends into an invalid range of memory.

UnsupportedPhysicalMemoryException when the underlying hardware does not support the given type, or when no matching `PhysicalMemoryTypeFilter`[32] has been registered with the `PhysicalMemoryManager`[33].

MemoryTypeConflictException when the specified base does not point to memory that matches the request type, or when `type` specifies incompatible memory attributes.

OutOfMemoryError when the requested type of memory exists, but there is not enough of it free to satisfy the request.

SecurityException when the application doesn't have permissions to access physical memory or the given range of memory.

RawMemoryAccess(Object, long, long)

Construct an instance of `RawMemoryAccess` with the given parameters, and set the object to the mapped state. When the platform supports virtual memory, map the raw memory into virtual memory.

The run time environment is allowed to choose the virtual address where the raw memory area corresponding to this object will be mapped. The attributes of the mapping operation are controlled by the vMFlags and vMAttributes of the `PhysicalMemoryTypeFilter` objects that matched this object's `type` parameter. (See `PhysicalMemoryTypeFilter.getVMAttributes`[34] and `PhysicalMemoryTypeFilter.getVMFlags`[35].

[32]Section 15.3.1.2
[33]Section 15.3.2.3
[34]Section 15.3.1.2.1
[35]Section 15.3.1.2.1

Deprecated since RTSJ version as of RTSJ 2.0. Use `RawMemoryFactory`[36] to create the appropriate `RawMemory`[37] object.

Signature

```
public
  RawMemoryAccess(Object type, long base, long size)
```

Parameters

type An instance of `Object` representing the type of memory required (e.g., *dma, shared*) - used to define the base address and control the mapping. When the required memory has more than one attribute, `type` may be an array of objects. When `type` is `null` or a reference to an array with no entries, any type of memory is acceptable. Note that `type` values are compared by reference (==), not by value (`equals`).

base The physical memory address of the region.

size The size of the area in bytes.

Throws

SecurityException when application doesn't have permissions to access physical memory, the specified range of addresses, or the given type of memory.

OffsetOutOfBoundsException when the address is invalid.

SizeOutOfBoundsException when the size is negative or extends into an invalid range of memory.

UnsupportedPhysicalMemoryException when the underlying hardware does not support the given type, or when no matching `PhysicalMemoryTypeFilter`[38] has been registered with the `PhysicalMemoryManager`[39].

MemoryTypeConflictException when the specified base does not point to memory that matches the request type, or when `type` specifies incompatible memory attributes.

OutOfMemoryError when the requested type of memory exists, but there is not enough of it free to satisfy the request.

15.3.2.5.2 Methods

---

[36] Section 12.4.3.5
[37] Section 12.4.1.16
[38] Section 15.3.1.2
[39] Section 15.3.2.3

getByte(long)

Gets the `byte` at the given offset in the memory area associated with this object. The byte is always loaded from memory in a single atomic operation.

Caching of the memory access is controlled by the memory `type` requested when the `RawMemoryAccess` instance was created. When 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.)
See Section RawMemoryAccess.map(long,long).

Signature

```
public
byte getByte(long offset)
```

Parameters

offset The offset in bytes from the beginning of the raw memory from which to load the byte.

Throws

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

OffsetOutOfBoundsException when the offset is negative or greater than the size of the raw memory area. The role of the `SizeOutOfBoundsException`[40] somewhat overlaps this exception since it is when the offset is within the object but outside the mapped area.

SecurityException when this access is not permitted by the security manager.

Returns

The byte from raw memory.

getBytes(long, byte[], int, int)

Gets `number` bytes starting at the given offset in the memory area associated with this object 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 memory `type` requested when the `RawMemoryAccess` instance was created. When the memory is not cached, this method guarantees serialized access (that is, the memory access at the memory

[40] Section 14.2.2.16

occurs in the same order as in the program. Multiple writes to the same location may not be coalesced.)
See Section RawMemoryAccess.map(long,long).

Signature

```
public
void getBytes(long offset, byte[] bytes, int low, int number)
```

Parameters

offset The offset in bytes from the beginning of the raw memory from which to start loading.
bytes The array into which the loaded items are placed.
low The offset which is the starting point in the given array for the loaded items to be placed.
number The number of items to load.

Throws

OffsetOutOfBoundsException when the offset is negative or greater than the size of the raw memory area. The role of the `SizeOutOfBoundsException`[41] somewhat overlaps this exception since it is when the offset is within the object but outside the mapped area.
SizeOutOfBoundsException when the object is not mapped, or when the byte 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 when the raw memory is unmapped or remapped mid-method.
ArrayIndexOutOfBoundsException when `low` is less than 0 or greater than `bytes.length - 1`, or when `low + number` is greater than or equal to `bytes.length`.
SecurityException when this access is not permitted by the security manager.

getInt(long)

Gets the `int` at the given offset in the memory area associated with this object. When the integer is aligned on a "natural" boundary it is always loaded from memory in a single atomic operation. When it is not on a natural boundary it may not be loaded atomically, and the number and order of the load operations is unspecified.

Caching of the memory access is controlled by the memory `type` requested when the `RawMemoryAccess` instance was created. When the memory is not cached, this method guarantees serialized access (that is, the memory access at the memory

[41] Section 14.2.2.16

occurs in the same order as in the program. Multiple writes to the same location may not be coalesced.)
See Section RawMemoryAccess.map(long,long).

Signature

```
public
int getInt(long offset)
```

Parameters

offset The offset in bytes from the beginning of the raw memory area from which to load the integer.

Throws

OffsetOutOfBoundsException when the offset is negative or greater than the size of the raw memory area. The role of the `SizeOutOfBoundsException`[42] somewhat overlaps this exception since it is when the offset is within the object but outside the mapped area.

SizeOutOfBoundsException when the object is not mapped, or when the integer falls in an invalid address range.

SecurityException when this access is not permitted by the security manager.

Returns

The integer from raw memory.

getInts(long, int[], int, int)

Gets `number` integers starting at the given offset in the memory area associated with this object and assign them to the int array passed starting at position `low`.

When the integers are aligned on natural boundaries each integer is loaded from memory in a single atomic operation. Groups of integers may be loaded together, but this is unspecified. When the integers are not aligned on natural boundaries they may not be loaded atomically and the number and order of load operations is unspecified.

Caching of the memory access is controlled by the memory `type` requested when the `RawMemoryAccess` instance was created. When 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.)
See Section RawMemoryAccess.map(long,long).

[42]Section 14.2.2.16

Signature

```
public
void getInts(long offset, int[] ints, int low, int number)
```

Parameters

offset The offset in bytes from the beginning of the raw memory area at which to start loading.

ints The array into which the integers read from the raw memory are placed.

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

number The number of integers to loaded.

Throws

OffsetOutOfBoundsException when the offset is negative or greater than the size of the raw memory area. The role of the `SizeOutOfBoundsException`[43] somewhat overlaps this exception since it is when the offset is within the object but outside the mapped area.

SizeOutOfBoundsException when the object is not mapped, or when the integers fall 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 `ints` array could, therefore, be partially updated when the raw memory is unmapped or remapped mid-method.

ArrayIndexOutOfBoundsException when `low` is less than 0 or greater than `bytes.length - 1`, or when `low + number` is greater than or equal to `bytes.length`.

SecurityException when this access is not permitted by the security manager.

getLong(long)

Gets the `long` at the given offset in the memory area associated with this object.

The load is not required to be atomic even it is located on a natural boundary.

Caching of the memory access is controlled by the memory `type` requested when the `RawMemoryAccess` instance was created. When 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.)

Signature

```
public
```

[43] Section 14.2.2.16

```
long getLong(long offset)
```

Parameters

offset The offset in bytes from the beginning of the raw memory area from which to load the long.

Throws

OffsetOutOfBoundsException when the offset is invalid.

SizeOutOfBoundsException when the object is not mapped, or when the double falls in an invalid address range.

SecurityException when this access is not permitted by the security manager.

Returns

The long from raw memory.

getLongs(long, long[], int, int)

Gets `number` longs starting at the given offset in the memory area associated with this object and assign them to the long array passed starting at position `low`.

The loads are not required to be atomic even when they are located on natural boundaries.

Caching of the memory access is controlled by the memory `type` requested when the `RawMemoryAccess` instance was created. When 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.)

See Section RawMemoryAccess.map(long,long).

Signature

```
public
void getLongs(long offset, long[] longs, int low, int number)
```

Parameters

offset The offset in bytes from the beginning of the raw memory area at which to start loading.

longs The array into which the loaded items are placed.

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

number The number of longs to load.

Throws

OffsetOutOfBoundsException when the offset is negative or greater than the

size of the raw memory area. The role of the `SizeOutOfBoundsException`[44] somewhat overlaps this exception since it is when the offset is within the object but outside the mapped area.
SizeOutOfBoundsException when the object is not mapped, or when a long 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 `longs` array could, therefore, be partially updated when the raw memory is unmapped or remapped mid-method.
ArrayIndexOutOfBoundsException when `low` is less than 0 or greater than `bytes.length - 1`, or when `low + number` is greater than or equal to `bytes.length`.
SecurityException when this access is not permitted by the security manager.

getMappedAddress

Gets the virtual memory location at which the memory region is mapped.
Deprecated since RTSJ version as of RTSJ 2.0 The program should never need this information.

Signature

```
public
long getMappedAddress()
```

Throws

IllegalStateException when the raw memory object is not in the mapped state.

Returns

The virtual address to which this is mapped (for reference purposes). Same as the base address when virtual memory is not supported.

getShort(long)

Gets the `short` at the given offset in the memory area associated with this object. When the short is aligned on a natural boundary it is always loaded from memory in a single atomic operation. When it is not on a natural boundary it may not be loaded atomically, and the number and order of the load operations is unspecified.

Caching of the memory access is controlled by the memory `type` requested when the `RawMemoryAccess` instance was created. When the memory is not cached, this method guarantees serialized access (that is, the memory access at the memory

[44]Section 14.2.2.16

occurs in the same order as in the program. Multiple writes to the same location may not be coalesced.)
See Section RawMemoryAccess.map(long,long).

Signature

`public`
`short getShort(long offset)`

Parameters

offset The offset in bytes from the beginning of the raw memory area from which to load the short.

Throws

OffsetOutOfBoundsException when the offset is negative or greater than the size of the raw memory area. The role of the `SizeOutOfBoundsException`[45] somewhat overlaps this exception since it is when the offset is within the object but outside the mapped area.

SizeOutOfBoundsException when the object is not mapped, or when the short falls in an invalid address range.

SecurityException when this access is not permitted by the security manager.

Returns

The short loaded from raw memory.

getShorts(long, short[], int, int)

Gets `number` shorts starting at the given offset in the memory area associated with this object and assign them to the short array passed starting at position `low`.

When the shorts are located on natural boundaries each short is loaded from memory in a single atomic operation. Groups of shorts may be loaded together, but this is unspecified.

When the shorts are not located on natural boundaries the load may not be atomic, and the number and order of load operations is unspecified. Caching of the memory access is controlled by the memory `type` requested when the `RawMemoryAccess` instance was created. When 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.)
See Section RawMemoryAccess.map(long,long).

[45]Section 14.2.2.16

Signature

```
public
void getShorts(long offset, short[] shorts, int low, int number)
```

Parameters

offset The offset in bytes from the beginning of the raw memory area from which to start loading.

shorts The array into which the loaded items are placed.

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

number The number of shorts to load.

Throws

OffsetOutOfBoundsException when the offset is negative or greater than the size of the raw memory area. The role of the `SizeOutOfBoundsException`[46] somewhat overlaps this exception since it is when the offset is within the object but outside the mapped area.

SizeOutOfBoundsException when the object is not mapped, or when a short 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 `shorts` array could, therefore, be partially updated when the raw memory is unmapped or remapped mid-method.

ArrayIndexOutOfBoundsException when `low` is less than 0 or greater than `bytes.length - 1`, or when `low + number` is greater than or equal to `bytes.length`.

SecurityException when this access is not permitted by the security manager.

map

Maps the physical memory range into virtual memory. No-op when the system doesn't support virtual memory.

The run time environment is allowed to choose the virtual address where the raw memory area corresponding to this object will be mapped. The attributes of the mapping operation are controlled by the vMFlags and vMAttributes of the `PhysicalMemoryTypeFilter` objects that matched this object's `type` parameter. (See `PhysicalMemoryTypeFilter.getVMAttributes`[47] and `PhysicalMemoryTypeFilter.getVMFlags`[48].

[46]Section 14.2.2.16
[47]Section 15.3.1.2.1
[48]Section 15.3.1.2.1

When the object is already mapped into virtual memory, this method does not change anything.

Signature

```
public
long map()
```

Throws

OutOfMemoryError when there is insufficient free virtual address space to map the object.

Returns

The starting point of the virtual memory range.

map(long)

Maps the physical memory range into virtual memory at the specified location. No-op when the system doesn't support virtual memory.

The attributes of the mapping operation are controlled by the vMFlags and vMAttributes of the `PhysicalMemoryTypeFilter` objects that matched this object's `type` parameter. (See `PhysicalMemoryTypeFilter.getVMAttributes`[49] and `PhysicalMemoryTypeFilter.getVMFlags`[50].

When the object is already mapped into virtual memory at a different address, this method remaps it to `base`.

When a remap is requested while another schedulable is accessing the raw memory, the map will block until one load or store completes. It can interrupt an array operation between entries.

Deprecated since RTSJ version as of RTSJ 2.0 No replacement

Signature

```
public
long map(long base)
```

Parameters

base The location to map at the virtual memory space.

Throws

OutOfMemoryError when there is insufficient free virtual memory at the specified address.

[49]Section 15.3.1.2.1

[50]Section 15.3.1.2.1

IllegalArgumentException when `base` is not a legal value for a virtual address, or the memory-mapping hardware cannot place the physical memory at the designated address.

Returns

The starting point of the virtual memory.

map(long, long)

Maps the physical memory range into virtual memory. No-op when the system doesn't support virtual memory.

The attributes of the mapping operation are controlled by the vMFlags and vMAttributes of the `PhysicalMemoryTypeFilter` objects that matched this object's `type` parameter. (See `PhysicalMemoryTypeFilter.getVMAttributes`[51] and `PhysicalMemoryTypeFilter.getVMFlags`[52].

When the object is already mapped into virtual memory at a different address, this method remaps it to `base`.

When a remap is requested while another schedulable is accessing the raw memory, the map will block until one load or store completes. It can interrupt an array operation between entries.

Deprecated since RTSJ version as of RTSJ 2.0 No replacement

Signature

```
public
long map(long base, long size)
```

Parameters

base The location to map at the virtual memory space.

size The size of the block to map in. When the size of the raw memory area is greater than `size`, the object is unchanged but accesses beyond the mapped region will throw `SizeOutOfBoundsException`[53]. When the size of the raw memory area is smaller than the mapped region access to the raw memory will behave as if the mapped region matched the raw memory area, but additional virtual address space will be consumed after the end of the raw memory area.

Throws

[51]Section 15.3.1.2.1
[52]Section 15.3.1.2.1
[53]Section 14.2.2.16

IllegalArgumentException when size is not greater than zero, `base` is not a legal value for a virtual address, or the memory-mapping hardware cannot place the physical memory at the designated address.

Returns

The starting point of the virtual memory.

setByte(long, byte)

Sets the `byte` at the given offset in the 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 memory `type` requested when the `RawMemoryAccess` instance was created. When 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.)

See Section RawMemoryAccess.map(long,long).

Signature

```
public
void setByte(long offset, byte value)
```

Parameters

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

value The byte to write.

Throws

OffsetOutOfBoundsException when the offset is negative or greater than the size of the raw memory area. The role of the `SizeOutOfBoundsException`[54] somewhat overlaps this exception since it is when the offset is within the object but outside the mapped area.

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

SecurityException when this access is not permitted by the security manager.

setBytes(long, byte[], int, int)

[54]Section 14.2.2.16

Sets `number` bytes starting at the given offset in the 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 memory `type` requested when the `RawMemoryAccess` instance was created. When 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.)
See Section RawMemoryAccess.map(long,long).

Signature

```
public
void setBytes(long offset, byte[] bytes, int low, int number)
```

Parameters

offset The offset in bytes from the beginning of the raw memory area to which to start writing.

bytes The array from which the items are obtained.

low The offset which is the starting point in the given array for the items to be obtained.

number The number of items to write.

Throws

OffsetOutOfBoundsException when the offset is negative or greater than the size of the raw memory area. The role of the `SizeOutOfBoundsException`[55] somewhat overlaps this exception since it is when the offset is within the object but outside the mapped area.

SizeOutOfBoundsException when the object is not mapped, or when the a short 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 store of the array into memory could, therefore, be only partially complete when the raw memory is unmapped or remapped mid-method.

ArrayIndexOutOfBoundsException when `low` is less than 0 or greater than `bytes.length - 1`, or when `low + number` is greater than or equal to `bytes.length`.

SecurityException when this access is not permitted by the security manager.

setInt(long, int)

[55]Section 14.2.2.16

Sets the `int` at the given offset in the memory area associated with this object. On most processor architectures an aligned integer can be stored in an atomic operation, but this is not required.

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 memory `type` requested when the `RawMemoryAccess` instance was created. When 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.)
See Section RawMemoryAccess.map(long,long).

Signature

```
public
void setInt(long offset, int value)
```

Parameters

offset The offset in bytes from the beginning of the raw memory area at which to write the integer.

value The integer to write.

Throws

OffsetOutOfBoundsException when the offset is negative or greater than the size of the raw memory area. The role of the `SizeOutOfBoundsException`[56] somewhat overlaps this exception since it is when the offset is within the object but outside the mapped area.

SizeOutOfBoundsException when the object is not mapped, or when the integer falls in an invalid address range.

SecurityException when this access is not permitted by the security manager.

setInts(long, int[], int, int)

Sets `number` ints starting at the given offset in the memory area associated with this object from the int array passed starting at position `low`. On most processor architectures each aligned integer can be stored in an atomic operation, but this is not required.

[56] Section 14.2.2.16

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 memory `type` requested when the `RawMemoryAccess` instance was created. When 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.)
See Section RawMemoryAccess.map(long,long).

Signature

```
public
void setInts(long offset, int[] ints, int low, int number)
```

Parameters

offset The offset in bytes from the beginning of the raw memory area at which to start writing.

ints The array from which the items are obtained.

low The offset which is the starting point in the given array for the items to be obtained.

number The number of items to write.

Throws

OffsetOutOfBoundsException when the offset is negative or greater than the size of the raw memory area. The role of the `SizeOutOfBoundsException`[57] somewhat overlaps this exception since it is when the offset is within the object but outside the mapped area.

SizeOutOfBoundsException when the object is not mapped, or when an int 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 store of the array into memory could, therefore, be only partially complete when the raw memory is unmapped or remapped mid-method.

ArrayIndexOutOfBoundsException when `low` is less than 0 or greater than `bytes.length - 1`, or when `low + number` is greater than or equal to `bytes.length`.

SecurityException when this access is not permitted by the security manager.

setLong(long, long)

[57]Section 14.2.2.16

Sets the `long` at the given offset in the memory area associated with this object. Even when it is aligned, the long value may not be updated atomically. It is unspecified how many load and store operations will be used or in what order.

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 memory `type` requested when the `RawMemoryAccess` instance was created. When 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.)
See Section RawMemoryAccess.map(long,long).

Signature

```
public
void setLong(long offset, long value)
```

Parameters

offset The offset in bytes from the beginning of the raw memory area at which to write the long.
value The long to write.

Throws

OffsetOutOfBoundsException when the offset is negative or greater than the size of the raw memory area. The role of the `SizeOutOfBoundsException`[58] somewhat overlaps this exception since it is when the offset is within the object but outside the mapped area.
SizeOutOfBoundsException when the object is not mapped, or when the long falls in an invalid address range.
SecurityException when this access is not permitted by the security manager.

setLongs(long, long[], int, int)

Sets `number` longs starting at the given offset in the memory area associated with this object from the long array passed starting at position `low`. Even when they are aligned, the long values may not be updated atomically. It is unspecified how many load and store operations will be used or in what order.

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

[58]Section 14.2.2.16

stored) in the presence of concurrent access.

Caching of the memory access is controlled by the memory `type` requested when the `RawMemoryAccess` instance was created. When 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.)

See Section RawMemoryAccess.map(long,long).

Signature

```
public
void setLongs(long offset, long[] longs, int low, int number)
```

Parameters

offset The offset in bytes from the beginning of the raw memory area at which to start writing.

longs The array from which the items are obtained.

low The offset which is the starting point in the given array for the items to be obtained.

number The number of items to write.

Throws

OffsetOutOfBoundsException when the offset is negative or greater than the size of the raw memory area. The role of the `SizeOutOfBoundsException`[59] somewhat overlaps this exception since it is when the offset is within the object but outside the mapped area.

SizeOutOfBoundsException when the object is not mapped, or when the a short 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 store of the array into memory could, therefore, be only partially complete when the raw memory is unmapped or remapped mid-method.

ArrayIndexOutOfBoundsException when `low` is less than 0 or greater than `bytes.length - 1`, or when `low + number` is greater than or equal to `bytes.length`.

SecurityException when this access is not permitted by the security manager.

setShort(long, short)

Sets the `short` at the given offset in the memory area associated with this object.

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

[59] Section 14.2.2.16

Caching of the memory access is controlled by the memory `type` requested when the `RawMemoryAccess` instance was created. When 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.)
See Section RawMemoryAccess.map(long,long).

Signature

```
public
void setShort(long offset, short value)
```

Parameters

offset The offset in bytes from the beginning of the raw memory area at which to write the short.

value The short to write.

Throws

OffsetOutOfBoundsException when the offset is negative or greater than the size of the raw memory area. The role of the `SizeOutOfBoundsException`[60] somewhat overlaps this exception since it is when the offset is within the object but outside the mapped area.

SizeOutOfBoundsException when the object is not mapped, or when the short falls in an invalid address range.

SecurityException when this access is not permitted by the security manager.

setShorts(long, short[], int, int)

Sets `number` shorts starting at the given offset in the memory area associated with this object from the short array passed starting at position `low`.

Each write of a short value may involve a load and a store, and it may have unspecified effects on surrounding shorts in the presence of concurrent access - even on other shorts in the array.

Caching of the memory access is controlled by the memory `type` requested when the `RawMemoryAccess` instance was created. When 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.)
See Section RawMemoryAccess.map(long,long).

[60] Section 14.2.2.16

Signature

```
public
void setShorts(long offset, short[] shorts, int low, int number)
```

Parameters

offset The offset in bytes from the beginning of the raw memory area at which to start writing.

shorts The array from which the items are obtained.

low The offset which is the starting point in the given array for the items to be obtained.

number The number of items to write.

Throws

OffsetOutOfBoundsException when the offset is negative or greater than the size of the raw memory area. The role of the `SizeOutOfBoundsException`[61] somewhat overlaps this exception since it is when the offset is within the object but outside the mapped area.

SizeOutOfBoundsException when the object is not mapped, or when the a short 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 store of the array into memory could, therefore, be only partially complete when the raw memory is unmapped or remapped mid-method.

ArrayIndexOutOfBoundsException when `low` is less than 0 or greater than `bytes.length - 1`, or when `low + number` is greater than or equal to `bytes.length`.

SecurityException when this access is not permitted by the security manager.

unmap

Unmap the physical memory range from virtual memory. This changes the raw memory from the mapped state to the unmapped state. When the platform supports virtual memory, this operation frees the virtual addresses used for the raw memory region.

When the object is already in the unmapped state, this method has no effect.

While a raw memory object is unmapped all attempts to set or get values in the raw memory will throw `SizeOutOfBoundsException`[62].

An unmapped raw memory object can be returned to mapped state with any of the object's `map` methods.

[61] Section 14.2.2.16

[62] Section 14.2.2.16

When an unmap is requested while another schedulable is accessing the raw memory, the `unmap` will throw an `IllegalStateException`. The `unmap` method can interrupt an array operation between entries.

Signature

```
public
void unmap()
```

15.3.2.6 RawMemoryFloatAccess

This class holds the accessor methods for accessing a raw memory area by float and double types. Implementations are required to implement this class when and only when the underlying Java Virtual Machine supports floating point data types.

See `RawMemoryAccess`[63] for commentary on changes in the preferred use of this class following RTSJ 2.0.

By default, the byte addressed by `offset` is the byte at the lowest address of the floating point processor's floating point representation. When the type of memory used for this `RawMemoryFloatAccess` region implements a non-standard floating point format, accessor methods in this class continue to select bytes starting at `offset` from the base address and continuing toward greater addresses. The memory type may control the mapping of these bytes into the primitive data type. The memory type could even select bytes that are not contiguous. In each case the documentation for the `PhysicalMemoryTypeFilter`[64] must document any mapping other than the "normal" one specified above.

All offset values used in this class are measured in bytes.

Atomic loads and stores on raw memory are defined in terms of physical memory. This memory may be accessible to threads outside the JVM and to non-programmed access (e.g., DMA), consequently atomic access must be supported by hardware. This specification is written with the assumption that all suitable hardware platforms support atomic loads for aligned floats. Atomic access beyond the specified minimum may be supported by the implementation.

Storing values into raw memory is more hardware-dependent than loading values. Many processor architectures do not support atomic stores of variables except for aligned stores of the processor's word size.

This class supports unaligned access to data, but it does not require the implementation to make such access atomic. Accesses to data aligned on its natural

[63]Section 15.3.2.5
[64]Section 15.3.1.2

boundary will be atomic when the processor implements atomic loads and stores of that data size.

Except where noted, accesses to raw memory are not atomic with respect to the memory or with respect to threads. A raw memory area could be updated by another thread, or even unmapped in the middle of a method.

The characteristics of raw-memory access are necessarily platform dependent. This specification provides a minimum requirement for the RTSJ platform, but it also supports a optional system properties that identify a platform's level of support for atomic raw put and get. (See `RawMemoryAccess`[65].) The properties represent a four-dimensional sparse array with boolean values whether that combination of access attributes is atomic. The default value for array entries is false.

Many of the constructors and methods in this class throw `OffsetOutOfBoundsException`[66]. This exception means that the value given in the offset parameter is either negative or outside the memory area.

Many of the constructors and methods in this class throw `SizeOutOfBoundsException`[67]. This exception means that the value given in the size parameter is either negative, larger than an allowable range, or would cause an accessor method to access an address outside of the memory area.

Deprecated since RTSJ version as of RTSJ 2.0. Use `RawMemory`[68].

Inheritance

java.lang.Object
 javax.realtime.RawMemoryAccess
  javax.realtime.RawMemoryFloatAccess

15.3.2.6.1 Constructors

RawMemoryFloatAccess(Object, long)

Construct an instance of `RawMemoryFloatAccess` with the given parameters, and set the object to the mapped state. When the platform supports virtual memory, map the raw memory into virtual memory.

The run time environment is allowed to choose the virtual address where the raw memory area corresponding to this object will be mapped. The attributes

[65] Section 15.3.2.5
[66] Section 14.2.2.12
[67] Section 14.2.2.16
[68] Section 12.4.1.16

of the mapping operation are controlled by the vMFlags and vMAttributes of the `PhysicalMemoryTypeFilter` objects that matched this object's `type` parameter. (See `PhysicalMemoryTypeFilter.getVMAttributes`[69] and `PhysicalMemoryTypeFilter.getVMFlags`[70].

Deprecated since RTSJ version as of RTSJ 2.0. Use `RawMemoryFactory.createRawFloat(RawMemoryRegion, long, int, int)`[71] or `RawMemoryFactory.createRawDouble(RawMemoryRegion, long, int, int)`[72].

Signature

```
public
  RawMemoryFloatAccess(Object type, long size)
```

Parameters

type An instance of `Object` representing the type of memory required (e.g., *dma, shared*) - used to define the base address and control the mapping. When the required memory has more than one attribute, `type` may be an array of objects. When `type` is `null` or a reference to an array with no entries, any type of memory is acceptable. Note that `type` values are compared by reference (==), not by value (`equals`).

size The size of the area in bytes.

Throws

SecurityException when the application doesn't have permissions to access physical memory, the specified range of addresses, or the given type of memory.

SizeOutOfBoundsException when the size is negative or extends into an invalid range of memory.

UnsupportedPhysicalMemoryException when the underlying hardware does not support the given type, or when no matching `PhysicalMemoryTypeFilter`[73] has been registered with the `PhysicalMemoryManager`[74].

MemoryTypeConflictException when the specified base does not point to memory that matches the request type, or when `type` specifies incompatible memory attributes.

[69] Section 15.3.1.2.1
[70] Section 15.3.1.2.1
[71] Section 12.4.3.5.3
[72] Section 12.4.3.5.3
[73] Section 15.3.1.2
[74] Section 15.3.2.3

OutOfMemoryError when the requested type of memory exists, but there is not enough of it free to satisfy the request.

RawMemoryFloatAccess(Object, long, long)

Construct an instance of `RawMemoryFloatAccess` with the given parameters, and set the object to the mapped state. When the platform supports virtual memory, map the raw memory into virtual memory.

The run time environment is allowed to choose the virtual address where the raw memory area corresponding to this object will be mapped. The attributes of the mapping operation are controlled by the vMFlags and vMAttributes of the `PhysicalMemoryTypeFilter` objects that matched this object's `type` parameter. (See `PhysicalMemoryTypeFilter.getVMAttributes`[75] and `PhysicalMemoryType-Filter.getVMFlags`[76].

Deprecated since RTSJ version as of RTSJ 2.0. Use `RawMemoryFactory.createRawFloat(RawMemo` `long, int, int)`[77] or `RawMemoryFactory.createRawDouble(RawMemoryRegion, long, int, int)`[78].

Signature

```
public
  RawMemoryFloatAccess(Object type, long base, long size)
```

Parameters

type An instance of `Object` representing the type of memory required (e.g., *dma, shared*) - used to define the base address and control the mapping. When the required memory has more than one attribute, `type` may be an array of objects. When `type` is `null` or a reference to an array with no entries, any type of memory is acceptable. Note that `type` values are compared by reference (==), not by value (`equals`).

base The physical memory address of the region.

size The size of the area in bytes.

Throws

[75]Section 15.3.1.2.1
[76]Section 15.3.1.2.1
[77]Section 12.4.3.5.3
[78]Section 12.4.3.5.3

SecurityException when the application doesn't have permissions to access physical memory, the specified range of addresses, or the given type of memory.
OffsetOutOfBoundsException when the address is invalid.
SizeOutOfBoundsException when the size is negative or extends into an invalid range of memory.
UnsupportedPhysicalMemoryException when the underlying hardware does not support the given type, or when no matching `PhysicalMemoryTypeFilter`[79] has been registered with the `PhysicalMemoryManager`[80].
MemoryTypeConflictException when the specified base does not point to memory that matches the request type, or when `type` specifies incompatible memory attributes.
OutOfMemoryError when the requested type of memory exists, but there is not

15.3.2.6.2 Methods

---

getDouble(long)

Gets the `double` at the given offset in the memory area associated with this object.

The load is not required to be atomic even it is located on a natural boundary.

Caching of the memory access is controlled by the memory `type` requested when the `RawMemoryAccess` instance was created. When 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.)

Signature

```
public
double getDouble(long offset)
```

Parameters

offset The offset in bytes from the beginning of the raw memory area from which to load the long.

Throws

OffsetOutOfBoundsException when the offset is invalid.

[79] Section 15.3.1.2
[80] Section 15.3.2.3

SizeOutOfBoundsException when the object is not mapped, or when the double falls in an invalid address range.

SecurityException when this access is not permitted by the security manager.

Returns

The double from raw memory.

getDoubles(long, double[], int, int)

Gets `number` doubles starting at the given offset in the memory area associated with this object and assign them to the double array passed starting at position `low`.

The loads are not required to be atomic even when they are located on natural boundaries.

Caching of the memory access is controlled by the memory `type` requested when the `RawMemoryAccess` instance was created. When 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.)

Signature

```
public
void getDoubles(long offset, double[] doubles, int low, int
number)
```

Parameters

offset The offset in bytes from the beginning of the raw memory area at which to start loading.

doubles The array into which the loaded items are placed.

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

number The number of doubles to load.

Throws

OffsetOutOfBoundsException when the offset is negative or greater than the size of the raw memory area. The role of the `SizeOutOfBoundsException` somewhat overlaps this exception since it is when the offset is within the object but outside the mapped area. (See `RawMemoryAccess.map(long,long)`[81]).

SizeOutOfBoundsException when the object is not mapped, or when a double 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

[81] Section 15.3.2.5.2

or remapped. The `doubles` array could, therefore, be partially updated when the raw memory is unmapped or remapped mid-method.
ArrayIndexOutOfBoundsException when `low` is less than 0 or greater than `bytes.length - 1`, or when `low + number` is greater than or equal to `bytes.length`.
SecurityException when this access is not permitted by the security manager.

getFloat(long)

Gets the `float` at the given offset in the memory area associated with this object. When the float is aligned on a "natural" boundary it is always loaded from memory in a single atomic operation. When it is not on a natural boundary it may not be loaded atomically, and the number and order of the load operations is unspecified.

Caching of the memory access is controlled by the memory `type` requested when the `RawMemoryAccess` instance was created. When 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.)

Signature

```
public
float getFloat(long offset)
```

Parameters

offset The offset in bytes from the beginning of the raw memory area from which to load the float.

Throws

OffsetOutOfBoundsException when the offset is negative or greater than the size of the raw memory area. The role of the `SizeOutOfBoundsException` somewhat overlaps this exception since it is when the offset is within the object but outside the mapped area. (See `RawMemoryAccess.map(long,long)`[82]).
SizeOutOfBoundsException when the object is not mapped, or when the float falls in an invalid address range.
SecurityException when this access is not permitted by the security manager.

Returns

The float from raw memory.

getFloats(long, float[], int, int)

[82]Section 15.3.2.5.2

Gets `number` floats starting at the given offset in the memory area associated with this object and assign them to the int array passed starting at position `low`.

When the floats are aligned on natural boundaries each float is loaded from memory in a single atomic operation. Groups of floats may be loaded together, but this is unspecified.

When the floats are not aligned on natural boundaries they may not be loaded atomically and the number and order of load operations is unspecified.

Caching of the memory access is controlled by the memory `type` requested when the `RawMemoryAccess` instance was created. When 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.)

Signature

```
public
void getFloats(long offset, float[] floats, int low, int number)
```

Parameters

offset The offset in bytes from the beginning of the raw memory area at which to start loading.

floats The array into which the floats loaded from the raw memory are placed.

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

number The number of floats to loaded.

Throws

OffsetOutOfBoundsException when the offset is negative or greater than the size of the raw memory area. The role of the `SizeOutOfBoundsException` somewhat overlaps this exception since it is when the offset is within the object but outside the mapped area. (See `RawMemoryAccess.map(long,long)`[83]).

SizeOutOfBoundsException when the object is not mapped, or when a float 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 `floats` array could, therefore, be partially updated when the raw memory is unmapped or remapped mid-method.

ArrayIndexOutOfBoundsException when `low` is less than 0 or greater than `bytes.length - 1`, or when `low + number` is greater than or equal to `bytes.length`.

SecurityException when this access is not permitted by the security manager.

setDouble(long, double)

[83]Section 15.3.2.5.2

Sets the `double` at the given offset in the memory area associated with this object. Even when it is aligned, the double value may not be updated atomically. It is unspecified how many load and store operations will be used or in what order.

Caching of the memory access is controlled by the memory `type` requested when the `RawMemoryAccess` instance was created. When 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.)

Signature

```
public
void setDouble(long offset, double value)
```

Parameters

offset The offset in bytes from the beginning of the raw memory area at which to write the double.

value The double to write.

Throws

OffsetOutOfBoundsException when the offset is negative or greater than the size of the raw memory area. The role of the `SizeOutOfBoundsException` somewhat overlaps this exception since it is when the offset is within the object but outside the mapped area. (See `RawMemoryAccess.map(long,long)`[84]).

SizeOutOfBoundsException when the object is not mapped, or when the double falls in an invalid address range.

SecurityException when this access is not permitted by the security manager.

setDoubles(long, double[], int, int)

Sets `number` doubles starting at the given offset in the memory area associated with this object from the double array passed starting at position `low`. Even when they are aligned, the double values may not be updated atomically. It is unspecified how many load and store operations will be used or in what order.

Caching of the memory access is controlled by the memory `type` requested when the `RawMemoryAccess` instance was created. When 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.)

[84]Section 15.3.2.5.2

Signature

```
public
void setDoubles(long offset, double[] doubles, int low, int
number)
```

Parameters

offset The offset in bytes from the beginning of the raw memory area at which to start writing.

doubles The array from which the items are obtained.

low The offset which is the starting point in the given array for the items to be obtained.

number The number of items to write.

Throws

OffsetOutOfBoundsException when the offset is negative or greater than the size of the raw memory area. The role of the `SizeOutOfBoundsException` somewhat overlaps this exception since it is when the offset is within the object but outside the mapped area. (See `RawMemoryAccess.map(long,long)`[85]).

SizeOutOfBoundsException when the object is not mapped, or when the a short 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 `doubles` array could, therefore, be partially updated when the raw memory is unmapped or remapped mid-method.

ArrayIndexOutOfBoundsException when `low` is less than 0 or greater than `bytes.length - 1`, or when `low + number` is greater than or equal to `bytes.length`.

SecurityException when this access is not permitted by the security manager.

setFloat(long, float)

Sets the `float` at the given offset in the memory area associated with this object. On most processor architectures an aligned float can be stored in an atomic operation, but this is not required.

Caching of the memory access is controlled by the memory `type` requested when the `RawMemoryAccess` instance was created. When 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.)

[85]Section 15.3.2.5.2

Signature

```
public
void setFloat(long offset, float value)
```

Parameters

offset The offset in bytes from the beginning of the raw memory area at which to write the integer.
value The float to write.

Throws

OffsetOutOfBoundsException when the offset is negative or greater than the size of the raw memory area. The role of the `SizeOutOfBoundsException` somewhat overlaps this exception since it is when the offset is within the object but outside the mapped area. (See `RawMemoryAccess.map(long,long)`[86]).
SizeOutOfBoundsException when the object is not mapped, or when the float falls in an invalid address range.
SecurityException when this access is not permitted by the security manager.

setFloats(long, float[], int, int)

Sets `number` floats starting at the given offset in the memory area associated with this object from the float array passed starting at position `low`. On most processor architectures each aligned float can be stored in an atomic operation, but this is not required. Caching of the memory access is controlled by the memory `type` requested when the `RawMemoryAccess` instance was created. When 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.)

Signature

```
public
void setFloats(long offset, float[] floats, int low, int number)
```

Parameters

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

Throws

[86] Section 15.3.2.5.2

OffsetOutOfBoundsException when the offset is negative or greater than the size of the raw memory area. The role of the `SizeOutOfBoundsException` somewhat overlaps this exception since it is when the offset is within the object but outside the mapped area. (See `RawMemoryAccess.map(long, long)`[87]).
SizeOutOfBoundsException when the object is not mapped, or when the float 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 store of the array into memory could, therefore, be only partially complete when the raw memory is unmapped or remapped mid-method.
ArrayIndexOutOfBoundsException when `low` is less than 0 or greater than `bytes.length - 1`, or when `low + number` is greater than or equal to `bytes.length`.
SecurityException when this access is not permitted by the security manager.

15.3.2.7 VTMemory

`VTMemory` is similar to `LTMemory`[88] except that the execution time of an allocation from a `VTMemory` area need not complete in linear time.

Methods from `VTMemory` should be overridden only by methods that use `super`.
Deprecated since RTSJ version as of RTSJ 2.0

Inheritance

java.lang.Object
  javax.realtime.MemoryArea
    javax.realtime.ScopedMemory
      javax.realtime.VTMemory

15.3.2.7.1 Constructors

VTMemory(long, long)

Creates a `VTMemory` with the given parameters.

Signature

[87] Section 15.3.2.5.2
[88] Section 11.4.3.4

```
public
  VTMemory(long initial, long maximum)
```

Parameters

initial The size in bytes of the memory to initially allocate for this area.
maximum The maximum size in bytes this memory area to which the size may grow.

Throws

IllegalArgumentException when `initial` is greater than `maximum` or when `initial` or `maximum` is less than zero.
OutOfMemoryError when there is insufficient memory for the `VTMemory` object or for the backing memory.

VTMemory(long, long, Runnable)

Creates a `VTMemory` with the given parameters.

Signature

```
public
  VTMemory(long initial, long maximum, Runnable logic)
```

Parameters

initial The size in bytes of the memory to initially allocate for this area.
maximum The maximum size in bytes this memory area to which the size may grow.
logic An instance of `Runnable` whose `run()` method will use `this` as its initial memory area. When `logic` is `null`, this constructor is equivalent to `VTMemory(long initial, long maximum)`[89].

Throws

IllegalArgumentException when `initial` is greater than `maximum`, or when `initial` or `maximum` is less than zero.
OutOfMemoryError when there is insufficient memory for the `VTMemory` object or for the backing memory.
IllegalAssignmentError when storing `logic` in `this` would violate the assignment rules.

[89] Section 15.3.2.7.1

VTMemory(SizeEstimator, SizeEstimator)

Creates a `VTMemory` with the given parameters.

Signature

```
public
  VTMemory(SizeEstimator initial, SizeEstimator maximum)
```

Parameters

initial The size in bytes of the memory to initially allocate for this area.
maximum The maximum size in bytes this memory area to which the size may grow estimated by an instance of `SizeEstimator`[90].

Throws

IllegalArgumentException when `initial` is `null`, `maximum` is `null`, `initial.getEstimate()` is greater than `maximum.getEstimate()`, or when `initial.getEstimate()` is less than zero.
OutOfMemoryError when there is insufficient memory for the `VTMemory` object or for the backing memory.

VTMemory(SizeEstimator, SizeEstimator, Runnable)

Creates a `VTMemory` with the given parameters.

Signature

```
public
  VTMemory(SizeEstimator initial, SizeEstimator maximum, Runnable logic)
```

Parameters

initial The size in bytes of the memory to initially allocate for this area.
maximum The maximum size in bytes this memory area to which the size may grow estimated by an instance of `SizeEstimator`[91].

[90]Section 11.4.3.12
[91]Section 11.4.3.12

logic An instance of `Runnable` whose `run()` method will use `this` as its initial memory area. When `logic` is `null`, this constructor is equivalent to `VTMemory(SizeEstimator initial, SizeEstimator maximum)`[92].

Throws

IllegalArgumentException when `initial` is `null`, `maximum` is `null`, `initial.getEstimate()` is greater than `maximum.getEstimate()`, or when `initial.getEstimate()` is less than zero.

OutOfMemoryError when there is insufficient memory for the `VTMemory` object or for the backing memory.

IllegalAssignmentError when storing `logic` in `this` would violate the assignment rules.

VTMemory(long)

Create an `VTMemory` of the given size. This constructor is equivalent to `VTMemory(size, size)`

Available since RTSJ 1.0.1

Signature

```
public
  VTMemory(long size)
```

Parameters

size The size in bytes of the memory to allocate for this area. This memory must be committed before the completion of the constructor.

Throws

IllegalArgumentException when `size` is less than zero.

OutOfMemoryError when there is insufficient memory for the `VTMemory` object or for the backing memory.

VTMemory(long, Runnable)

Create an `VTMemory` of the given size. This constructor is equivalent to `VTMemory(size, size, logic)`.

[92]Section 15.3.2.7.1

Available since RTSJ 1.0.1

Signature

```
public
  VTMemory(long size, Runnable logic)
```

Parameters

size The size in bytes of the memory to allocate for this area. This memory must be committed before the completion of the constructor.

logic The `run()` of the given `Runnable` will be executed using `this` as its initial memory area. When `logic` is `null`, this constructor is equivalent to `VTMemory(long size)`[93].

Throws

IllegalArgumentException when `size` is less than zero.

OutOfMemoryError when there is insufficient memory for the `VTMemory` object or for the backing memory.

IllegalAssignmentError when storing `logic` in `this` would violate the assignment rules.

VTMemory(SizeEstimator)

Create an `VTMemory` of the given size. This constructor is equivalent to `VTMemory(size, size)`.

Available since RTSJ 1.0.1

Signature

```
public
  VTMemory(SizeEstimator size)
```

Parameters

size An instance of `SizeEstimator`[94] used to give an estimate of the initial size. This memory must be committed before the completion of the constructor.

[93]Section 15.3.2.7.1
[94]Section 11.4.3.12

Throws

IllegalArgumentException when `size` is `null`, or `size.getEstimate()` is less than zero.
OutOfMemoryError when there is insufficient memory for the `VTMemory` object or for the backing memory.

VTMemory(SizeEstimator, Runnable)

Create an `VTMemory` of the given size.
Available since RTSJ 1.0.1

Signature

```
public
  VTMemory(SizeEstimator size, Runnable logic)
```

Parameters

size An instance of `SizeEstimator`[95] used to give an estimate of the initial size. This memory must be committed before the completion of the constructor.
logic The `run()` of the given `Runnable` will be executed using `this` as its initial memory area. When `logic` is `null`, this constructor is equivalent to `VTMemory(SizeEstimator initial)`[96].

Throws

IllegalArgumentException when `size` is `null`, or `size.getEstimate()` is less than zero.
OutOfMemoryError when there is insufficient memory for the `VTMemory` object or for the backing memory.
IllegalAssignmentError when storing `logic` in `this` would violate the assignment rules.

15.3.2.7.2 Methods

---

[95]Section 11.4.3.12
[96]Section 15.3.2.7.1

toString

Create a string representing this object. The string is of the form
`(VTMemory) Scoped memory # num`
where `num` uniquely identifies the `VTMemory` area.

Signature

```
public
java.lang.String toString()
```

Returns

A string representing the value of `this`.

15.3.2.8 VTPhysicalMemory

An instance of `VTPhysicalMemory` allows objects to be allocated from a range of physical memory with particular attributes, determined by their memory type. This memory area has the same semantics as `ScopedMemory`[97] memory areas, and the same performance restrictions as `VTMemory`.

No provision is made for sharing object in `VTPhysicalMemory` with entities outside the JVM that creates them, and, while the memory backing an instance of `VTPhysicalMemory` could be shared by multiple JVMs, the class does not support such sharing.

Methods from `VTPhysicalMemory` should be overridden only by methods that use `super`.

See Section MemoryArea

See Section ScopedMemory

See Section VTMemory

See Section LTMemory

See Section LTPhysicalMemory

See Section ImmortalPhysicalMemory

[97]Section 11.4.3.11

See Section RealtimeThread

See Section NoHeapRealtimeThread

Deprecated since RTSJ version as of RTSJ 2.0

Inheritance

java.lang.Object
  javax.realtime.MemoryArea
    javax.realtime.ScopedMemory
      javax.realtime.VTPhysicalMemory

15.3.2.8.1 Constructors

VTPhysicalMemory(Object, long)

Create an instance of `VTPhysicalMemory` with the given parameters.
See Section PhysicalMemoryManager

Signature

```
public
  VTPhysicalMemory(Object type, long size)
```

Parameters

type An instance of `Object` representing the type of memory required (e.g., *dma, shared*) - used to define the base address and control the mapping. When the required memory has more than one attribute, `type` may be an array of objects. When `type` is `null` or a reference to an array with no entries, any type of memory is acceptable. Note that `type` values are compared by reference (==), not by value (`equals`).

size The size of the area in bytes.

Throws

SecurityException when the application doesn't have permissions to access physical memory or the given range of memory.

SizeOutOfBoundsException when the implementation detects that `size` extends beyond physically addressable memory.
UnsupportedPhysicalMemoryException when the underlying hardware does not support the given type, or when no matching `PhysicalMemoryTypeFilter`[98] has been registered with the `PhysicalMemoryManager`[99].
MemoryTypeConflictException when the specified base does not point to memory that matches the requested type, or when `type` specifies incompatible memory attributes.
IllegalArgumentException when `size` is less than zero.

VTPhysicalMemory(Object, long, long)

Create an instance of `VTPhysicalMemory` with the given parameters.
See Section PhysicalMemoryManager

Signature

```
public
  VTPhysicalMemory(Object type, long base, long size)
```

Parameters

type An instance of `Object` representing the type of memory required (e.g., *dma, shared*) - used to define the base address and control the mapping. When the required memory has more than one attribute, `type` may be an array of objects. When `type` is `null` or a reference to an array with no entries, any type of memory is acceptable. Note that `type` values are compared by reference (==), not by value (`equals`).
base The physical memory address of the area.
size The size of the area in bytes.

Throws

SecurityException when the application doesn't have permissions to access physical memory or the given range of memory.
SizeOutOfBoundsException when the implementation detects that `size` extends beyond physically addressable memory.
OffsetOutOfBoundsException when the `base` address is invalid.

[98]Section 15.3.1.2
[99]Section 15.3.2.3

UnsupportedPhysicalMemoryException when the underlying hardware does not support the given type, or when no matching `PhysicalMemoryTypeFilter`[100] has been registered with the `PhysicalMemoryManager`[101].
MemoryTypeConflictException when the specified base does not point to memory that matches the requested type, or when `type` specifies incompatible memory attributes.
MemoryInUseException when the specified memory is already in use.

VTPhysicalMemory(Object, SizeEstimator)

Create an instance of `VTPhysicalMemory` with the given parameters.
See Section PhysicalMemoryManager

Signature

```
public
  VTPhysicalMemory(Object type, SizeEstimator size)
```

Parameters

type An instance of `Object` representing the type of memory required (e.g., *dma, shared*) - used to define the base address and control the mapping. When the required memory has more than one attribute, `type` may be an array of objects. When `type` is `null` or a reference to an array with no entries, any type of memory is acceptable. Note that `type` values are compared by reference (==), not by value (`equals`).
size A size estimator for this area.

Throws

SecurityException when the application doesn't have permissions to access physical memory or the given range of memory.
SizeOutOfBoundsException when the implementation detects that the size estimate from `size` extends beyond physically addressable memory.
UnsupportedPhysicalMemoryException when the underlying hardware does not support the given type, or when no matching `PhysicalMemoryTypeFilter`[102] has been registered with the `PhysicalMemoryManager`[103].

[100] Section 15.3.1.2
[101] Section 15.3.2.3
[102] Section 15.3.1.2
[103] Section 15.3.2.3

MemoryTypeConflictException when the specified base does not point to memory that matches the requested type, or when `type` specifies incompatible memory attributes.
IllegalArgumentException when `size` is `null`, or `size.getEstimate()` is negative.

VTPhysicalMemory(Object, long, SizeEstimator)

Create an instance of `VTPhysicalMemory` with the given parameters.
See Section PhysicalMemoryManager

Signature

```
public
  VTPhysicalMemory(Object type, long base, SizeEstimator size)
```

Parameters

type An instance of `Object` representing the type of memory required (e.g., *dma, shared*) - used to define the base address and control the mapping. When the required memory has more than one attribute, `type` may be an array of objects. When `type` is `null` or a reference to an array with no entries, any type of memory is acceptable. Note that `type` values are compared by reference (==), not by value (`equals`).
base The physical memory address of the area.
size A size estimator for this memory area.

Throws

SecurityException when the application doesn't have permissions to access physical memory or the given range of memory.
SizeOutOfBoundsException when the implementation detects that the size estimate from `size` extends beyond physically addressable memory.
OffsetOutOfBoundsException when the `base` address is invalid.
UnsupportedPhysicalMemoryException when the underlying hardware does not support the given type, or when no matching `PhysicalMemoryTypeFilter`[104] has been registered with the `PhysicalMemoryManager`[105].

[104] Section 15.3.1.2
[105] Section 15.3.2.3

MemoryTypeConflictException when the specified base does not point to memory that matches the requested type, or when `type` specifies incompatible memory attributes.
MemoryInUseException when the specified memory is already in use.
IllegalArgumentException when `size` is `null`, or `size.getEstimate()` is negative.

VTPhysicalMemory(Object, long, Runnable)

Create an instance of `VTPhysicalMemory` with the given parameters.
See Section PhysicalMemoryManager

Signature

```
public
  VTPhysicalMemory(Object type, long size, Runnable logic)
```

Parameters

type An instance of `Object` representing the type of memory required (e.g., *dma, shared*) - used to define the base address and control the mapping. When the required memory has more than one attribute, `type` may be an array of objects. When `type` is `null` or a reference to an array with no entries, any type of memory is acceptable. Note that `type` values are compared by reference (==), not by value (`equals`).
size The size of the area in bytes.
logic The `run()` method of this object will be called whenever `MemoryArea.enter()`[106] is called. When `logic` is `null`, `logic` must be supplied when the memory area is entered.

Throws

SecurityException when the application doesn't have permissions to access physical memory or the given range of memory.
SizeOutOfBoundsException when the implementation detects that `size` extends beyond physically addressable memory.
UnsupportedPhysicalMemoryException when the underlying hardware does not support the given type, or when no matching `PhysicalMemoryTypeFilter`[107] has been registered with the `PhysicalMemoryManager`[108].

[106] Section 11.4.3.6.2
[107] Section 15.3.1.2
[108] Section 15.3.2.3

MemoryTypeConflictException when the specified base does not point to memory that matches the requested type, or when `type` specifies incompatible memory attributes.
IllegalAssignmentError when storing `logic` in `this` would violate the assignment rules.

VTPhysicalMemory(Object, long, long, Runnable)

Create an instance of `VTPhysicalMemory` with the given parameters.
See Section PhysicalMemoryManager

Signature

```
public
  VTPhysicalMemory(Object type, long base, long size, Runnable logic)
```

Parameters

type An instance of `Object` representing the type of memory required (e.g., *dma, shared*) - used to define the base address and control the mapping. When the required memory has more than one attribute, `type` may be an array of objects. When `type` is `null` or a reference to an array with no entries, any type of memory is acceptable. Note that `type` values are compared by reference (==), not by value (`equals`).
base The physical memory address of the area.
size The size of the area in bytes.
logic The `run()` method of this object will be called whenever `MemoryArea.enter()`[109] is called. When `logic` is `null`, `logic` must be supplied when the memory area is entered.

Throws

SizeOutOfBoundsException when the implementation detects that `size` extends beyond physically addressable memory.
SecurityException when the application doesn't have permissions to access physical memory or the given range of memory.
OffsetOutOfBoundsException when the `base` address is invalid.
UnsupportedPhysicalMemoryException when the underlying hardware does not support the given type, or when no matching `PhysicalMemoryTypeFilter`[110]

[109] Section 11.4.3.6.2
[110] Section 15.3.1.2

has been registered with the `PhysicalMemoryManager`[111].
MemoryTypeConflictException when the specified base does not point to memory that matches the requested type, or when `type` specifies incompatible memory attributes.
MemoryInUseException when the specified memory is already in use.
IllegalAssignmentError when storing `logic` in `this` would violate the assignment rules.

VTPhysicalMemory(Object, SizeEstimator, Runnable)

Create an instance of `VTPhysicalMemory` with the given parameters.
See Section PhysicalMemoryManager

Signature

```
public
  VTPhysicalMemory(Object type, SizeEstimator size, Runnable logic)
```

Parameters

type An instance of `Object` representing the type of memory required (e.g., *dma, shared*) - used to define the base address and control the mapping. When the required memory has more than one attribute, `type` may be an array of objects. When `type` is `null` or a reference to an array with no entries, any type of memory is acceptable. Note that `type` values are compared by reference (==), not by value (`equals`).
size A size estimator for this area.
logic The `run()` method of this object will be called whenever `MemoryArea.enter()`[112] is called. When `logic` is `null`, `logic` must be supplied when the memory area is entered.

Throws

SecurityException when the application doesn't have permissions to access physical memory or the given range of memory.
SizeOutOfBoundsException when the implementation detects that the size estimate from `size` extends beyond physically addressable memory.
UnsupportedPhysicalMemoryException when the underlying hardware does not support the given type, or when no matching `PhysicalMemoryTypeFilter`[113]

[111] Section 15.3.2.3
[112] Section 11.4.3.6.2
[113] Section 15.3.1.2

has been registered with the `PhysicalMemoryManager`[114].
MemoryTypeConflictException when the specified base does not point to memory that matches the requested type, or when `type` specifies incompatible memory attributes.
IllegalArgumentException when `size` is `null`, or `size.getEstimate()` is negative.
IllegalAssignmentError when storing `logic` in `this` would violate the assignment rules.

VTPhysicalMemory(Object, long, SizeEstimator, Runnable)

Create an instance of `VTPhysicalMemory` with the given parameters.
See Section PhysicalMemoryManager

Signature

```
public
  VTPhysicalMemory(Object type, long base, SizeEstimator size, Runnable logic)
```

Parameters

type An instance of `Object` representing the type of memory required (e.g., *dma, shared*) - used to define the base address and control the mapping. When the required memory has more than one attribute, `type` may be an array of objects. When `type` is `null` or a reference to an array with no entries, any type of memory is acceptable. Note that `type` values are compared by reference (==), not by value (`equals`).
base The physical memory address of the area.
size A size estimator for this memory area.
logic The `run()` method of this object will be called whenever `MemoryArea.enter()`[115] is called. When `logic` is `null`, `logic` must be supplied when the memory area is entered.

Throws

SecurityException when the application doesn't have permissions to access physical memory or the given range of memory.

[114]Section 15.3.2.3
[115]Section 11.4.3.6.2

SizeOutOfBoundsException when the implementation detects that the size estimate from `size` extends beyond physically addressable memory.
OffsetOutOfBoundsException when the `base` address is invalid.
UnsupportedPhysicalMemoryException when the underlying hardware does not support the given type, or when no matching `PhysicalMemoryTypeFilter`[116] has been registered with the `PhysicalMemoryManager`[117].
MemoryTypeConflictException when the specified base does not point to memory that matches the requested type, or when `type` specifies incompatible memory attributes.
MemoryInUseException when the specified memory is already in use.
IllegalArgumentException when `size` is `null`, or `size.getEstimate()` is negative.
IllegalAssignmentError when storing `logic` in `this` would violate the assignment rules.

VTPhysicalMemory(PhysicalMemoryName, long)

Create an instance of `VTPhysicalMemory` with the given parameters.
See Section PhysicalMemoryManager

Available since RTSJ 2.0

Signature

```
public
  VTPhysicalMemory(PhysicalMemoryName type, long size)
```

```
throws SecurityException, SizeOutOfBoundsException,
UnsupportedPhysicalMemoryException, MemoryTypeConflictException
```

Parameters

type Used to define the base address and control the mapping. When `type` is `null` any type of memory is acceptable. Note that `type` values are compared by reference (==), not by value (`equals`).
size The size of the area in bytes.

Throws

[116]Section 15.3.1.2
[117]Section 15.3.2.3

SecurityException The application doesn't have permissions to access physical memory or the given type of memory.
SizeOutOfBoundsException `size` is negative or extends into an invalid range of memory.
UnsupportedPhysicalMemoryException when the underlying hardware does not support `type`.
MemoryTypeConflictException The specified base does not point to memory that matches the request type, or when `type` specifies attributes with a conflict.

VTPhysicalMemory(PhysicalMemoryName, long, long)

Create an instance of `VTPhysicalMemory` with the given parameters.
See Section PhysicalMemoryManager

Available since RTSJ 2.0

Signature

```
public
  VTPhysicalMemory(PhysicalMemoryName type, long base, long size)


 throws SecurityException, SizeOutOfBoundsException,
 OffsetOutOfBoundsException, UnsupportedPhysicalMemoryException,
 MemoryTypeConflictException, MemoryInUseException
```

Parameters

type Used to define the base address and control the mapping. When `type` is `null` any type of memory is acceptable. Note that `type` values are compared by reference (==), not by value (`equals`).
base The physical memory address of the area.
size The size of the area in bytes.

Throws

SecurityException The application doesn't have permissions to access physical memory or the given range of memory.
OffsetOutOfBoundsException The address is invalid.
SizeOutOfBoundsException `size` is negative or extends into an invalid range of memory.
UnsupportedPhysicalMemoryException when the underlying hardware does not support `type`.

MemoryTypeConflictException when `base` does not point to memory that matches `type`, or when `type` specifies conflicting attributes.
MemoryInUseException The specified memory is already in use.

VTPhysicalMemory(PhysicalMemoryName, SizeEstimator)

Create an instance of `VTPhysicalMemory` with the given parameters.
See Section PhysicalMemoryManager

Available since RTSJ 2.0

Signature

```
public
  VTPhysicalMemory(PhysicalMemoryName type, SizeEstimator size)
```

```
throws SecurityException, SizeOutOfBoundsException,
UnsupportedPhysicalMemoryException, MemoryTypeConflictException
```

Parameters

type Used to define the base address and control the mapping. When `type` is `null` any type of memory is acceptable. Note that `type` values are compared by reference (==), not by value (`equals`).
size A size estimator for this area.

Throws

SecurityException The application doesn't have permissions to access physical memory or the given type of memory.
SizeOutOfBoundsException `size` is negative or extends into an invalid range of memory.
UnsupportedPhysicalMemoryException when the underlying hardware does not support `type`.
MemoryTypeConflictException when `base` does not point to memory that matches `type`, or when `type` specifies conflicting attributes.

VTPhysicalMemory(PhysicalMemoryName, long, SizeEstimator)

Create an instance of `VTPhysicalMemory` with the given parameters.

See Section PhysicalMemoryManager

Available since RTSJ 2.0

Signature

```
public
  VTPhysicalMemory(PhysicalMemoryName type, long base, SizeEstimator size)

 throws SecurityException, SizeOutOfBoundsException,
 OffsetOutOfBoundsException, UnsupportedPhysicalMemoryException,
 MemoryTypeConflictException, MemoryInUseException
```

Parameters

type Used to define the base address and control the mapping. When `type` is `null` any type of memory is acceptable. Note that `type` values are compared by reference (==), not by value (`equals`).

base The physical memory address of the area.

size A size estimator for this memory area.

Throws

SecurityException The application doesn't have permissions to access physical memory or the given range of memory.

OffsetOutOfBoundsException The address is invalid.

SizeOutOfBoundsException `size` is negative or extends into an invalid range of memory.

UnsupportedPhysicalMemoryException when the underlying hardware does not support `type`.

MemoryTypeConflictException when `base` does not point to memory that matches `type`, or when `type` specifies conflicting attributes.

MemoryInUseException The specified memory is already in use.

VTPhysicalMemory(PhysicalMemoryName, long, Runnable)

Create an instance of `VTPhysicalMemory` with the given parameters.

See Section PhysicalMemoryManager

Available since RTSJ 2.0

Signature

```
public
  VTPhysicalMemory(PhysicalMemoryName type, long size, Runnable logic)
```

```
throws SecurityException, SizeOutOfBoundsException,
UnsupportedPhysicalMemoryException, MemoryTypeConflictException
```

Parameters

type Used to define the base address and control the mapping. When `type` is `null` any type of memory is acceptable. Note that `type` values are compared by reference (==), not by value (`equals`).

logic `enter` this memory area with this `Runnable` after the memory area is created.

Throws

SecurityException The application doesn't have permissions to access physical memory or the given type of memory.

SizeOutOfBoundsException `size` is negative or extends into an invalid range of memory.

UnsupportedPhysicalMemoryException when the underlying hardware does not support `type`.

MemoryTypeConflictException when `base` does not point to memory that matches `type`, or when `type` specifies conflicting attributes.

VTPhysicalMemory(PhysicalMemoryName, long, long, Runnable)

Create an instance of `VTPhysicalMemory` with the given parameters.

See Section PhysicalMemoryManager

Available since RTSJ 2.0

Signature

```
public
  VTPhysicalMemory(PhysicalMemoryName type, long base, long size, Runnable logi
```

```
throws SecurityException, SizeOutOfBoundsException,
OffsetOutOfBoundsException, UnsupportedPhysicalMemoryException,
MemoryTypeConflictException, MemoryInUseException
```

Parameters

type Used to define the base address and control the mapping. When `type` is `null` any type of memory is acceptable. Note that `type` values are compared by reference (==), not by value (`equals`).
base The physical memory address of the area.
size The size of the area in bytes.
logic `enter` this memory area with this `Runnable` after the memory area is created.

Throws

SecurityException The application doesn't have permissions to access physical memory or the given range of memory.
OffsetOutOfBoundsException The address is invalid.
SizeOutOfBoundsException `size` is negative or extends into an invalid range of memory.
UnsupportedPhysicalMemoryException when the underlying hardware does not support `type`.
MemoryTypeConflictException when `base` does not point to memory that matches `type`, or when `type` specifies conflicting attributes.
MemoryInUseException The specified memory is already in use.

VTPhysicalMemory(PhysicalMemoryName, SizeEstimator, Runnable)

Create an instance of `VTPhysicalMemory` with the given parameters.
See Section PhysicalMemoryManager

Available since RTSJ 2.0

Signature

```
public
  VTPhysicalMemory(PhysicalMemoryName type, SizeEstimator size, Runnable logic)

 throws SecurityException, SizeOutOfBoundsException,
 UnsupportedPhysicalMemoryException, MemoryTypeConflictException
```

Parameters

type Used to define the base address and control the mapping. When `type` is `null` any type of memory is acceptable. Note that `type` values are compared by reference (==), not by value (`equals`).

size A size estimator for this area.

logic `enter` this memory area with this `Runnable` after the memory area is created.

Throws

SecurityException The application doesn't have permissions to access physical memory or the given type of memory.

SizeOutOfBoundsException `size` is negative or extends into an invalid range of memory.

UnsupportedPhysicalMemoryException when the underlying hardware does not support `type`.

MemoryTypeConflictException when `base` does not point to memory that matches `type`, or when `type` specifies conflicting attributes.

VTPhysicalMemory(PhysicalMemoryName, long, SizeEstimator, Runnable)

Create an instance of `VTPhysicalMemory` with the given parameters.
See Section PhysicalMemoryManager

Available since RTSJ 2.0

Signature

```
public
  VTPhysicalMemory(PhysicalMemoryName type, long base, SizeEstimator size, Runn

 throws SecurityException, SizeOutOfBoundsException,
 OffsetOutOfBoundsException, UnsupportedPhysicalMemoryException,
 MemoryTypeConflictException, MemoryInUseException
```

Parameters

type Used to define the base address and control the mapping. When `type` is `null` any type of memory is acceptable. Note that `type` values are compared by reference (==), not by value (`equals`).

base The physical memory address of the area.

size A size estimator for this memory area.
logic `enter` this memory area with this `Runnable` after the memory area is created.

Throws

SecurityException The application doesn't have permissions to access physical memory or the given range of memory.
OffsetOutOfBoundsException The address is invalid.
SizeOutOfBoundsException `size` is negative or extends into an invalid range of memory.
UnsupportedPhysicalMemoryException when the underlying hardware does not support `type`.
MemoryTypeConflictException when `base` does not point to memory that matches `type`, or when `type` specifies conflicting attributes.
MemoryInUseException The specified memory is already in use.

15.3.2.8.2 Methods

toString

Creates a string representing this object. The string is of the form `(VTPhysicalMemory) Scoped memory # num` where `num` is a number that uniquely identifies this `VTPhysicalMemory` memory area.

Signature

```
public
java.lang.String toString()
```

Returns

A string representing the value of `this`.

15.4 Rationale

These are interface that have been shown to be less the ideal. They have been replaced by classes that better fulfill the requirements. Compatibility can be provided by implemenations that use existing facilities so there is not reason to continue requiring their inclusion new implementations.

Chapter 16

Conformance, Compliance, and Portability

16.1 Minimum Implementations

The flexibility of the `RTSJ` indicates that implementations may provide different semantics for scheduling, synchronization, and garbage collection. This section defines what minimum semantics for these areas and other semantics and APIs required of all implementations of the `RTSJ`. Other than what is described in the optional modules or explicitly noted as optionally required, the `RTSJ` does not allow any subsetting of the APIs in the `javax.realtime` package; however, some of the classes are specific to certain well-known scheduling or synchronization algorithms and may have no underlying support in a minimum implementation of the `RTSJ`. The `RTSJ` provides these classes as standard parent classes for implementations supporting such algorithms.

16.2 Modules

As described in Section 3.2, the `RTSJ` now has modules. Every implementation must implement the Core module. If any of the other modules is provided, it must be provided in full. None of the classes of an unimplemented module should be present.

16.3 Optionally Required Components

The `RTSJ` does not, in general, support the concept of optional components of the specification. Optional components would further complicate the already difficult task of writing WORA (Write Once Run Anywhere) software components for realtime systems. However, understanding the difficulty of providing implementations of mechanisms for which there is no underlying support, the RTSJ does provide for a few exceptions. Any components that are considered optional will be listed as such in the class definitions.

The most notable optional component of the specification is the POSIXSignalHandler. A conformant implementation which implements the Device module must support POSIX signals if and only if the underlying system supports them.

16.3.1 Deployment Implementation

The minimum scheduling semantics that must be supported in all implementations of the RTSJ are fixed-priority preemptive scheduling and at least 28 unique priority levels. By fixed-priority we mean that the system does not change the priority of any RealtimeThread or NoHeapRealtimeThread except, temporarily, for priority inversion avoidance. Note, however, that application code may change such priorities. What the RTSJ precludes by this statement is scheduling algorithms that change thread priorities according to policies for optimizing throughput (such as increasing the priority of threads that have been receiving few processor cycles because of higher priority threads (aging)). The 28 unique priority levels are required to be unique to preclude implementations from using fewer priority levels of underlying systems to implement the required 28 by simplistic algorithms (such as lumping four RTSJ priorities into seven buckets for an underlying system that only supports seven priority levels). It is sufficient for systems with fewer than 28 priority levels to use more sophisticated algorithms to implement the required 28 unique levels as long as RealtimeThreads and NoHeapRealtimeThreads behave as though there were at least 28 unique levels. (e.g. if there were 28 RealtimeThreads ($t_1, ..., t_{28}$) with priorities ($p_1, ..., p_{28}$), respectively, where the value of p_1 was the highest priority and the value of p_2 the next highest priority, etc., then for all executions of threads t_1 through t_{28} thread t_1 would *always* execute in preference to threads $t_2, ..., t_{28}$ and thread t_2 would *always* execute in preference to threads $t_3, ..., t_{28}$, etc.)

The minimum synchronization semantics that must be supported in all deployment implementations of the RTSJ are detailed in the above section on synchronization and repeated here.

All deployment implementations of the RTSJ must provide an implementation of the synchronized primitive with default behavior that ensures that there is no unbounded priority inversion. Furthermore, this must apply to code if it is run within the implementation as well as to realtime threads. The priority inheritance protocol must be implemented by default.

All threads waiting to acquire a resource must be queued in priority order. This applies to the processor as well as to synchronized blocks. If threads with the same exact priority are possible under the active scheduling policy, threads with the same priority are queued in FIFO order. (Note that these requirements apply only to the required base scheduling policy and hence use the specific term ”priority”). In particular:

- Threads waiting to enter synchronized blocks are granted access to the synchronized block in priority order.
- A blocked thread that becomes ready to run is given access to the processor in priority order.
- A thread whose execution eligibility is explicitly set by itself or another thread is given access to the processor in priority order.
- A thread that performs a `yield()` will be given access to the processor after waiting threads of the same priority.
- However, threads that are preempted in favor of a thread with higher priority may be given access to the processor at any time as determined by a particular implementation. The implementation is required to provide documentation stating exactly the algorithm used for granting such access.

The `RTSJ` does not require any particular garbage collection algorithm; however, every deployment implementation must either support memory areas or have a realtime garbage collection. In the later case, the realtime limitations must be documented. All implementations of the `RTSJ` must support the class `GarbageCollector` and implement all of its methods.

16.4 Simulation Implementation

An implementation that chooses not to provide realtime guarentees, is termed a simulation implementation. Such an implementation does not need to provide the realtime characteristic described above, but does need to at least provide all the APIs of the core module. A simulation implementation can be a production system, but not for realtime applications. This enables a conventional JVM to make the base APIs available to a wider audience without changing its performance characteristics.

16.5 Documentation Requirements

In order to properly engineer a realtime system, an understanding of the cost associated with any arbitrary code segment is required. This is especially important for operations that are performed by the runtime system, largely hidden from the programmer. (An example of this is the maximum expected latency before the garbage collector can be interrupted.)

The `RTSJ` does not require specific performance or latency numbers to be matched. Rather, to be conformant to this specification, an implementation must provide documentation regarding the expected behavior of particular mechanisms. The mechanisms requiring such documentation, and the specific data to be provided, will be detailed in the class and method definitions.

Appendix A

Bibliography

[1] *Portable Operating System Interface (POSIX®) Part 1: System Application Program Interface, International Standard ISO/IEC 9945-1, 1996 (E) IEEE Std 1003.1*, 1996 edition ed. The Institute of Electrical and Electronics Engineers, Inc., 1996.

[2] Barr, M. Memory types. *Embedded Systems Programming* (2001), 103–104.

[3] Burns, A., and Wellings, A. J. *Real-Time Systems and Programming Languages:*, 4th ed. Addison Wesley, 2010.

[4] Dos Santos, O. M., and Wellings, A. Cost enforcement in the real-time specification for java. *Real-Time Syst. 37*, 2 (Nov. 2007), 139–179.

[5] Gosling, J., Joy, B., Steele, G., Bracha, G., and Buckley, A. *The Java Language Specification Java SE 8 Edition.* Oracle, 2014.

[6] Lindholm, T., Yellin, F., Bracha, G., and Buckley, A. *The Java Virtual Machine Specification Java SE 8 Edition.* Oracle, 2014.

[7] Regehr, J. Safe and structured use of interrupts in real-time and embedded software. In *Handbook of Real-Time and Embedded Systems*, I. Lee, J. Y.-T. Leug, and S. H. Son, Eds. Chapman and Hall/CRC, 2007, pp. 16–1–16–12.